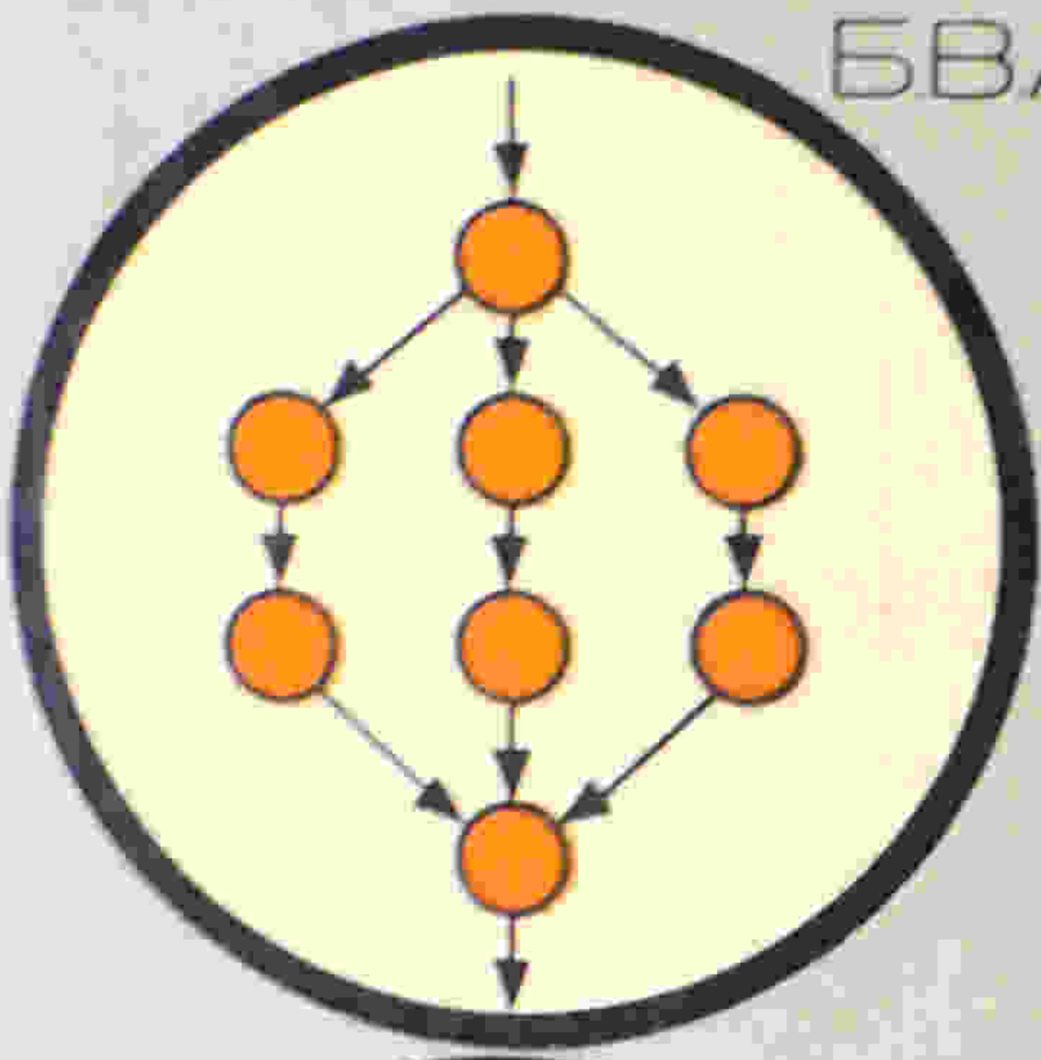
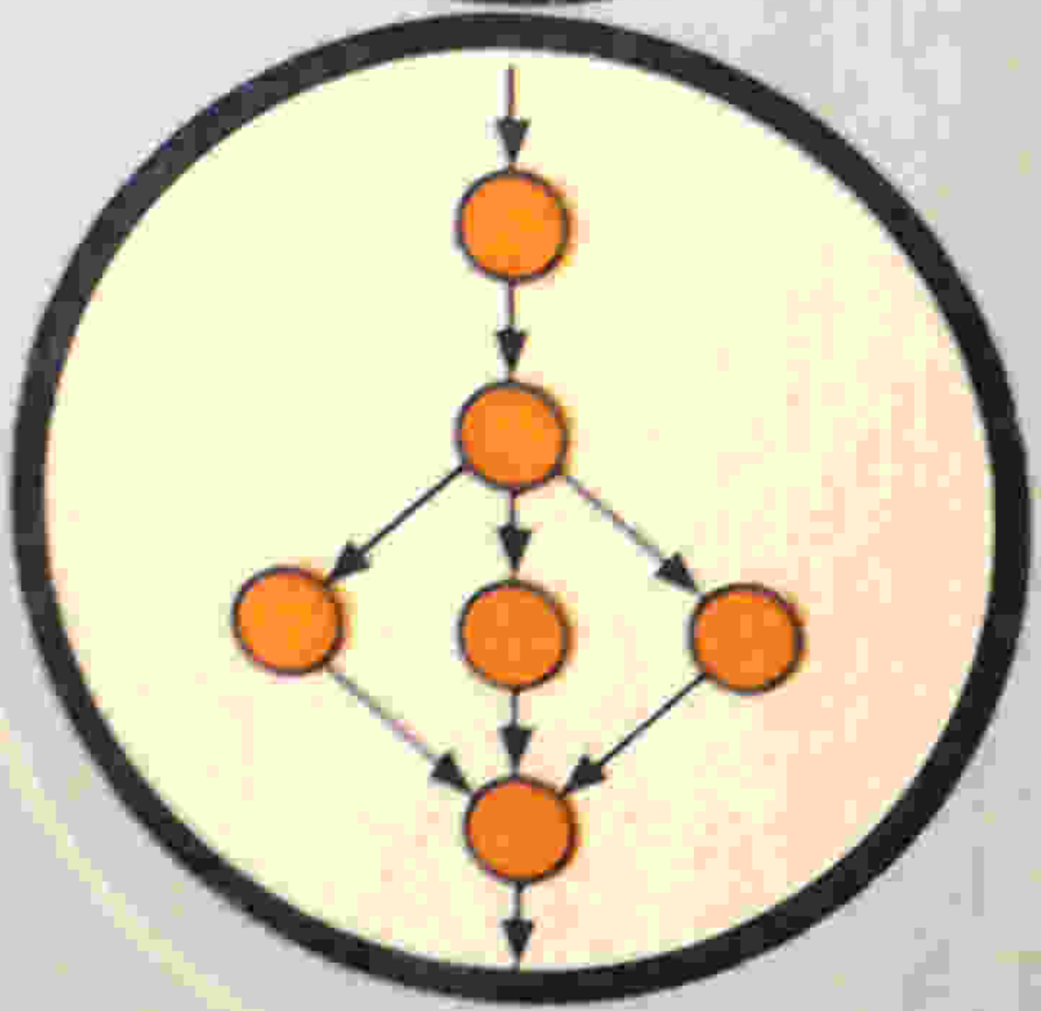
СИСТЕМЫ ОПТИМИЗАЦИИ ПРОГРАММ





Б.В.Архангельский
А.И.Никитин

КИЕВ
„ТЕХНІКА"
1983

Б.В.Архангельский А.И.Никитин

СИСТЕМЫ ОПТИМИЗАЦИИ ПРОГРАММ

Киев
«Техніка»
1983

32.97
А87

Архангельский Б. В., Никитин А. И.
А87 Системы оптимизации программ.— К.: Техніка, 1983.— 167 с., ил.— Библиогр.: с. 162—166.
В пер.: 90 к. 4000 экз.
В книге впервые в отечественной литературе систематизированы вопросы разработки и создания систем оптимизации программ. Изложены основные подходы к решению задачи оптимизации программ, указаны уровни языков, на которых выполняются оптимизационные преобразования, области оптимизации и структуры современных систем оптимизации программ. Рассчитана на инженерно-технических работников-специалистов по системному программированию, а также может быть полезна студентам вузов соответствующих специальностей.

А $\frac{2405000000-139}{М202(04)-83}$ 70.83 32.97

Рецензенты кандидаты техн. наук В. В. Литвинов, В. П. Рубцов
Редакция литературы по энергетике, электронике, кибернетике и связи
Зав. редакцией З. В. Божко

© Издательство «Техніка», 1983

Архангельский Б. В., Никитин А. И.

А87 Системы оптимизации программ.— К.: Техніка, 1983.— 167 с., ил.— Библиогр.: с. 162—166.

В пер.: 90 к. 4000 экз.

В книге впервые в отечественной литературе систематизированы вопросы разработки и создания систем оптимизации программ. Изложены основные подходы к решению задачи оптимизации программ, указаны уровни языков, на которых выполняются оптимизационные преобразования, области оптимизации и структуры современных систем оптимизации программ. Рассчитана на инженерно-технических работников-специалистов по системному программированию, а также может быть полезна студентам вузов соответствующих специальностей

А $\frac{2405000000-139}{М202(04)-83}$ 70.83 32.97

Рецензенты кандидаты техн. наук В. В. Литвинов, В. П. Рубцов
Редакция литературы по энергетике, электронике, кибернетике и связи
Зав. редакцией З. В. Божко

ВВЕДЕНИЕ

Задача оптимизации программы — получить программу, реализующую заданный содержательный алгоритм и минимизирующую при этом использование какого-либо ресурса ЭВМ, в первую очередь, времени процессора и/или основной памяти. В настоящее время практически для всех распространенных языков программирования созданы оптимизирующие трансляторы. Эти трансляторы повсеместно используются математиками (в первую очередь, для создания программ реального времени), что доказывает необходимость их существования наряду с быстро работающими трансляторами, генерирующими программы невысокого качества.

Созданные в последние годы оптимизирующие трансляторы существенно отличаются от первых оптимизирующих трансляторов, созданных в начале шестидесятых годов. Эти отличия, первоначально имевшие количественный характер, с развитием теории и практики программирования приобрели качественный характер, который заключается в следующем.

1. Значительно возросла сложность реализуемых оптимизационных процедур, в частности, увеличились области оптимизации, что привело к качественно иным методам реализации этих процедур, к необходимости выполнения сложного анализа управляющего и информационного графов программы, к использованию таких специальных приемов, как, например, факторизация программы. Усложнение оптимизационных процедур привело также к изменению характера алгоритмов их реализации. Если в первых оптимизирующих трансляторах алгоритмы реализации оптимизационных процедур являлись неотъемлемой частью алгоритмов трансляции, то в настоящее время эти алгоритмы, как правило, реализуются отдельным блоком, причем зачастую имеется возможность управления работой этого блока. При таком подходе процессы трансляции, оптимизации и генерации объектного кода отделяются друг от друга, собственно транслятор и его оптимизирующий блок являются независимыми, но дополняющими друг друга частями одной системы программирования.

2. Значительно расширился собственно набор оптимизационных процедур, включающий процедуры, выполняющие глобальную оптимизацию.

3. Увеличилась сложность оптимизационных процедур и расширился их набор и, как следствие, получили развитие методы анализа и преобразования управляющего и информационного графов программы, что позволило более полно изучить свойства и особенности реальных программ. Это привело к пониманию того, что различные этапы процесса создания программ (трансляция, отладка и верификация, модификация, документирование и сопровождение) глубоко связаны между собой. Поэтому в создаваемые оптимизирующие трансляторы разработчики начали включать дополнительные средства, в частности средства отладки.

Появление указанных отличий привело к тому, что современный оптимизирующий транслятор представляет собой сложную систему автоматизации программирования, в которой процесс трансляции является хотя и одним из основных процессов, но не доминирующим. Более того, в настоящее время созданы оптимизирующие трансляторы на уровне входных языков (например, ФОРТРАНа). Основная функция этих трансляторов — выполнение оптимизации программ (sourse to sourse optimisation), у которых отсутствует фаза трансляции программ в обычном понимании этого слова, так как уровень языка после их преобразования не меняется. На основании этого, по мнению авторов, такие системы целесообразно называть системами оптимизации программ (а не оптимизирующими трансляторами), так как этот термин более точно отражает их сущность *.

Увеличение объема программ и, следовательно, усложнение их ограничивают возможности человека, не позволяют программисту учитывать далекие связи в программе, чтобы осуществлять эффективную ручную оптимизацию даже на уровне языка ассемблера (психологические эксперименты показывают, что человек способен сконцентрироваться одновременно не более чем на 5—11 разных объектах). По этим же причинам отпала необходимость в отдельных программистах, производящих идеальный по качеству программный продукт значительного объема, не требующий дополнительной оптимизации. Качество программ в настоящее время повышают с помощью систем оптимизации, являющихся неотъемлемым компонентом современной технологии производства программ.

Созданные к настоящему времени лучшие системы оптимизации программ могут производить объектные программы для сложных исходных программ на ФОРТРАНе или АЛГОЛе, по качеству почти не уступающие программам, написанным на языке ассемблера программистами высокой квалификации, но с гораздо меньшими затратами. Более того, опыт создания математического обеспечения показывает, что для сохранения конкурен-

* Термин оптимизирующий транслятор применяется (для того чтобы избежать путаницы) только при ссылках на работы других авторов.

тоспособности любая современная операционная система должна включать в себя, как правило, системы оптимизации программ с различными степенями оптимизации. Примером может служить ОС ЕС ЭВМ, включающая в себя системы оптимизации программ ФОРТРАН-Н и ПЛ/1-Н. Эта точка зрения нашла свое подтверждение и в прогнозах, согласно которым универсальный оптимизатор будет являться неотъемлемой частью математического обеспечения ЭВМ четвертого поколения.

В книге не рассмотрены вопросы оптимизации при параллельном программировании, так как программа, оптимальная для однопроцессорных ЭВМ, может быть неоптимальной для многопроцессорных ЭВМ. Это определяет то, что методы оптимизации при параллельном программировании существенно отличаются от методов оптимизации программ для однопроцессорных ЭВМ, и более того, составляют самостоятельную и большую область исследований. Рассмотреть все специфические особенности этой области исследований в небольшом разделе не представляется возможным. Предполагается, что читатели настоящей книги владеют программированием на языках ФОРТРАН, ПЛ/1, а также знакомы с основными понятиями теории графов.

ГЛАВА 1

ЗАДАЧА ОПТИМИЗАЦИИ ПРОГРАММ

1. ОБЩИЕ ПОНЯТИЯ

К общим понятиям относятся: критерии оптимизации; теоретические и практические аспекты оптимизации; набор и стратегия применения оптимизационных процедур; оптимизационный эффект процедур.

Критерии оптимизации. Качество программ является достаточно сложным понятием, не всегда формализуемым. Основной критерий, отражающий качество программ, зависит от двух параметров: времени выполнения программы и объема используемой ею основной памяти. В общем случае в качестве критерия оптимизации можно выбрать достаточно сложную функцию от указанных двух параметров. Однако, как правило, используют частные критерии, т. е. отдельно время выполнения программы и отдельно объем используемой ею памяти, и под оптимизацией понимают преобразование, уменьшающее один из этих частных критериев. Таким образом, задача полной оптимизации формулируется следующим образом: преобразовать программу так, чтобы она выполнялась как можно быстрее (или так, чтобы она использовала возможно меньший объем памяти), выполняя те же функции, что и до преобразования.

Если объем памяти, используемый программой,— постоянная, четко определенная и легко вычисляемая величина, то время выполнения программы не является постоянным, так как зависит от значений начальных данных. При этом максимальное время выполнения программы может отличаться от минимального в десятки раз. Максимальное время выполнения для ряда специальных программ (например, программ реального времени), как наиболее важную величину выбирают в качестве критерия оптимизации. Если для большинства наборов начальных значений время выполнения программы незначительно отличается от некоторой средней величины, то в качестве критерия выбирают средневзвешенное время выполнения программы с учетом вероятности появления того или иного набора начальных данных.

Два частных критерия оптимизации — время выполнения программы и объем используемой ею памяти, в общем случае, противоречат друг другу, так как сокращения времени выполнения программы, как правило, можно добиться за счет увеличе-

ния объема используемой памяти и наоборот. В этом случае при выборе критерия вступают в силу эвристические соображения программиста, отдающего предпочтение одному из них.

Теоретические и практические аспекты оптимизации. В общем случае не существует алгоритма полной оптимизации, т. е. нахождения самой компактной или самой быстрой программы, выполняющей содержательно те же функции, что и некоторая исходная программа (отображающей одинаковым образом входные данные в выходные). Исключение составляют линейные программы, т. е. программы, имеющие один вход и один выход, с последовательным выполнением операторов. Для таких программ при некоторых дополнительных ограничениях возможна полная оптимизация, т. е. можно построить полную систему эквивалентных преобразований и разработать алгоритм построения оптимальной программы [14]. Однако линейные программы составляют лишь узкий класс реальных программ, и решение для них задачи оптимизации не позволяет говорить о сколько-нибудь исчерпывающем решении задачи. Поэтому термин «оптимизация программ» является в общем случае неверным. Более правильным был бы термин «улучшение программ». Тем не менее исторически широкое распространение получил термин «оптимизация программ», который в настоящее время является общепризнанным и будет использоваться в дальнейшем изложении.

Отсутствие алгоритма полной оптимизации ничуть не снижает практического значения задачи оптимизации и, более того, не ограничивает области теоретических исследований. Это объясняется тем, что большинство реальных программ образуют с теоретической точки зрения достаточно узкий класс, для которого заведомо можно получить конкретные теоретические и практические результаты. Для многих разработанных к настоящему времени алгоритмов оптимизации можно специально построить программы, применение к которым этих алгоритмов дает отрицательный результат. Однако это никак не снижает теоретической и практической ценности алгоритмов, а только в разумных пределах ограничивает область их применения, в частности, такой областью является, как правило, большинство реальных программ.

В настоящей книге рассмотрены практические аспекты задачи оптимизации программ, т. е. нахождения алгоритмов преобразований, которые, как правило, приводят к уменьшению соответствующего критерия оптимизации для данной программы. Именно такой подход к задаче оптимизации развивается и совершенствуется многими авторами.

Набор и стратегия применения оптимизационных процедур. Система оптимизации программ включает в себя набор оптимизационных процедур. Каждая из этих процедур производит равносильные преобразования оптимизируемой программы (или некоторого ее участка) и, в общем случае, несколько повышает ее

качество, т. е. частично оптимизирует. Создание системы оптимизации требует прежде всего выбора множества оптимизационных процедур, определения для каждой из них условий применения, а также нахождения эффективной стратегии их выполнения с учетом влияния преобразований, производимых процедурами, друг на друга [83]. Набор оптимизационных процедур существенно зависит от класса задач, для которых предназначена разрабатываемая система, от языка программирования, внутреннего языка, операционной системы и особенностей используемой ЭВМ. Фиксация набора оптимизационных процедур позволяет говорить об оптимальной программе как о наилучшей по отношению к выбранному набору. Будем называть такую программу относительно оптимальной.

После выбора множества оптимизационных процедур решается задача определения стратегии выполнения этих процедур на основе учета условий их применения и взаимных влияний между процедурами. Само множество оптимизационных процедур уже определяет в некоторой степени последовательность их выполнения. Действительно, выполнение оптимизационных процедур над некоторым участком программы меняет этот участок и одновременно формирует некоторые конкретные условия, определяющие возможность выполнения других процедур. Например, выполнение константных действий [8] над оператором $A = 3.0 + 6.7$ формирует константные условия для выполнения над этим оператором процедуры подстановки [12].

Рассмотрим две оптимизационные процедуры A и B. Пусть после применения процедуры A происходит такое изменение участка программы или контекстных условий для B, что применение B, возможного для исходного участка программы, становится невозможным для результирующего. Тогда говорят, что оптимизационная процедура A обладает тупиковостью (тупикова) по отношению к оптимизационной процедуре B. Пусть, наоборот, после применения оптимизационной процедуры A происходит такое изменение участка программы или контекстных условий для B, что применение B, невозможного для исходного участка программы, становится возможным для результирующего. Тогда говорят, что оптимизационная процедура A обладает повторностью (повторна) по отношению к оптимизационной процедуре B. Например, процедура подстановки [12] повторна по отношению к процедуре выполнения константных действий [8].

Программа, полученная в результате некоторой стратегии применения полного набора оптимизационных процедур и не содержащая возможности дальнейшего их применения, но отличающаяся от относительно оптимальной, называется тупиковой по отношению к данному набору. Алгоритм (стратегия) применения оптимизационных процедур, дающий тупиковую программу, называется тупиковым, а относительно оптимальную программу, минимальным. Для построения минимального и тупикового алгоритмов большое значение имеет изучение тупиковости и повторности, однако эта задача зачастую является достаточно слож-

ной. Следует заметить, что тупиковая программа по своим параметрам, характеризующим ее качество, может быть достаточно далека от относительно оптимальной программы.

Наиболее полным решением задачи оптимизации программ является выбор эффективного набора оптимизационных процедур и поиск минимального (по отношению к этому набору) алгоритма. Однако на практике часто используют не минимальный, а некоторый тупиковый алгоритм, достаточно близкий к минимальному. Это объясняется трудностями, возникающими при практической реализации минимальных алгоритмов, обусловленных в основном нехваткой ресурсов ЭВМ. Поэтому при практической реализации рассматриваемых алгоритмов возникает дополнительная задача оценки «близости» некоторого тупикового алгоритма к минимальному.

Требования, предъявляемые к оптимизационным процедурам. Оптимизационная процедура должна: 1) сохранять функции, которые выполняет программа, т. е. производить равносильное преобразование; 2) не ухудшать качество ни одной программы, входящей в область применения этой процедуры; 3) улучшать качество хотя бы некоторых программ, входящих в область применения этой процедуры.

Первое требование означает, что преобразованная оптимизационными процедурами программа должна быть равносильна исходной При этом, вообще говоря, эта равносильность может соблюдаться только на некоторых наборах начальных значений. На других наборах начальных значений, которые не интересуют программиста, эта равносильность может нарушаться и, более того, преобразованная программа может быть неправильной с семантической точки зрения.

Второе требование, как это следует из сказанного выше, сводится к описанию области применения оптимизационных процедур. Как правило, под областью применения для большинства процедур по умолчанию подразумевается класс реальных программ.

В соответствии с третьим требованием выполняемые процедуры должны действительно оптимизировать программу, т. е. должно существовать множество программ из области применения процедур, преобразование которых этими процедурами улучшает их. Следует отметить, что всегда существуют программы, преобразование которых оптимизационными процедурами не повышает их качества. В частности, к таким программам относятся тупиковые по отношению к данному набору.

Следует заметить, что на практике допускается ослабление первого требования. В частности, оптимизирующие преобразования могут дополнять программу. Это выражается в том, что если программа A определена на наборе данных S, то программа $T(A)$, полученная в результате применения преобразования T к

программе A, определена на наборе данных S_1 более широком, чем S, причем результаты работы программ A и $T(A)$ на наборе данных S совпадают [90]. На практике, например, алгоритм процедуры чистки циклов может выполнять преобразования, нарушающие смысл некоторых исходных программ [82]. Поэтому формулирование оптимизационной процедуры означает одновременное формулирование контекстных условий, при соблюдении которых выполнение процедуры приводит к программе, равносильной исходной.

Оптимизационный эффект процедур. Результат преобразования участка программы некоторой оптимизационной процедурой характеризуется разницей между значениями критериев оптимизации исходного и результирующего участков программы. Эта разница называется оптимизационным эффектом процедуры. Согласно данному определению, оптимизационный эффект процедуры зависит только от структуры преобразуемого участка и не зависит от контекста, в котором этот участок находится. В отличие от него повышение качества всей программы (т. е. полный эффект), получаемое в результате применения оптимизационной процедуры, зависит и от контекста, в котором находится область оптимизации. Это обусловлено, в частности, различной частотой выполнения областей оптимизации при выполнении программы. Мерой повышения качества программы в целом является функция от оптимизационного эффекта процедуры и частоты выполнения области оптимизации. Эта мера и определяет эффективность применения процедуры, которая должна учитываться отдельно в каждом конкретном случае. Если некоторое преобразование снижает качество программы, то говорят, что оно имеет отрицательный оптимизационный эффект в отличие от положительного оптимизационного эффекта, отражающего повышение качества программ. В частности, оптимизационные процедуры могут иметь отрицательный оптимизационный эффект по отношению к участкам, не принадлежащим к области применения этих процедур. Поэтому для каждой оптимизационной процедуры существенное значение имеет определение ее области применения. В гл. 2.1 при описании одной из самых известных процедур чистки циклов приведен пример, когда применение этой процедуры может привести к отрицательному оптимизационному эффекту.

Помимо рассматриваемого оптимизационного эффекта, называемого прямым оптимизационным эффектом, так как его можно определить сразу же после выполнения процедуры, существует еще косвенный оптимизационный эффект. Суть его заключается в том, что любые преобразования программы, в том числе и преобразования, выполняемые оптимизационными процедурами, могут влиять на оптимизационный эффект выполняемых вслед за ними процедур. Это влияние называют косвенным оптимизационным эффектом процедуры. В некоторых случаях косвенный

оптимизационный эффект процедуры может быть более существенным, чем прямой.

Влияние оптимизации на другие этапы создания программ. Характерной чертой оптимизации программ является то, что результаты ее выполнения (сокращение времени счета и объема используемой памяти) реализуются только на этапе работы программы. Однако выполнение оптимизационных преобразований изменяет исходный текст программы, что, как правило, ухудшает ее читаемость и понимаемость. Поэтому, несмотря на то, что результаты выполнения оптимизации ориентированы только на этапы работы программы, они влияют также и на другие этапы создания программы, такие, как отладка, модификация, верификация и документация, для которых читаемость и понимаемость программы имеют важное значение (следует отметить, что это влияние, как правило, носит негативный характер). Учет этого влияния при реализации алгоритмов оптимизационных процедур породил требование — выполнять только самые необходимые, минимальные преобразования. Однако это не решает полностью вопроса. Сохранить эффективность этапа оптимизации программ и одновременно минимизировать влияние выполняемых преобразований на другие этапы создания программ можно только в случае, если оптимизация следует за всеми остальными этапами создания программ и непосредственно предшествует ее работе. Такой подход позволяет также минимизировать ресурсы ЭВМ, необходимые для осуществления собственно оптимизации, так как в этом случае она выполняется только один раз над полностью готовой к работе программой, и ее не приходится осуществлять каждый раз после очередного вносимого в программу изменения. Таким образом, следует придерживаться следующего правила: выполнять оптимизацию только для полностью отлаженной программы, готовой к выполнению, и никогда не оптимизировать программу только ради оптимизации. Исключение составляет специальный класс программ, к которым предъявляются очень высокие требования по надежности. Для таких программ во избежание различных непредвиденных ситуаций при их разработке, в частности при модификации и оптимизации, соответствующими ГОСТами жестко закрепляется требование, чтобы непосредственно процессу исполнения этих программ предшествовал процесс их тестирования и отладки. Другими словами, процесс тестирования и отладки является последним этапом в процессе создания таких программ.

В последние годы широкое распространение получил структурный подход к программированию [92], одной из основных целей которого является получение легко читаемых и понимаемых программ. Приемы, с помощью которых достигается эта легкость чтения и понимаемости программ [39], находятся в противоречии

с оптимальностью программ и приводят в частности, к снижению их качества в смысле времени выполнения и объема используемой памяти. Однако, если выполнять оптимизацию для полностью отлаженной и готовой к счету программы, то можно, создавая структурированные программы, а следовательно, сохраняя все преимущества структурного подхода к программированию, повышать качество этих программ. То же самое можно сказать про другие специальные приемы создания программ, такие, как метод формализованных технических заданий [28] или метод последовательного уточнения программы [118].

Кроме того, следует придерживаться такого правила: оптимизацию осуществлять в автоматическом или автоматизированном режиме. Иногда у программистов, особенно тех, кто программирует многие годы, возникает желание вручную создать сложную и вместе с тем высокоэффективную программу. Однако это желание является следствием недооценки сложности анализа управляющих и информационных связей программы, с которым связано выполнение элементарных, на первый взгляд, оптимизационных преобразований. Поэтому зачастую такая ручная оптимизация приводит к изменению смысла программы и даже потере ее эффективности. Кроме того, в соревновании с компилятором программист заранее обречен на неудачу, так как ЭВМ выполняет подобного рода преобразования гораздо лучше, чем человек, и такое соревнование программисту просто не нужно.

Сравнение эффективности оптимизационных процедур. Наиболее полная проверка эффективности алгоритмов оптимизационных процедур — это их реализация и длительное практическое использование; причем эффективность как отдельной процедуры, так и их совокупности характеризуется повышением качества преобразуемых программ и временем выполнения самих процедур.

Для большинства программ (и в первую очередь для программ, выполняемых в реальном масштабе времени на малых ЭВМ) выигрыш, обусловленный повышением качества преобразуемых программ, более важен, чем проигрыш, обусловленный увеличением времени трансляции. В большинстве созданных систем оптимизации процедуры реализуются, как правило, в виде единого блока, и поэтому можно оценить эффективность только комплекса процедур, а не отдельной процедуры. К настоящему времени создано достаточно большое число систем оптимизации программ как у нас в стране [2, 80, 98], так и за рубежом [103, 116]. В основу этих систем заложены разные принципы, реализованы достаточно различные оптимизационные процедуры, да и алгоритмы реализации одних и тех же процедур существенно отличаются друг от друга. Поэтому представляют большой интерес оценка абсолютной эффективности работы системы и сравнение эффективности различных систем.

Наиболее распространенный метод оценки абсолютной эффективности работы системы оптимизации заключается в сравнении параметров порождаемых ею программ в объектных кодах с аналогичными параметрами программ в объектных кодах, составленных опытными программистами. И те и другие программы должны, естественно, реализовать один и тот же алгоритм решения задачи. Применение понятия абсолютной оценки при таком подходе основано на предположении, что опытный программист пишет оптимальные программы (хотя для программ большого объема это предположение, как правило, неверно). Метод обладает рядом недостатков: он трудоемок (каждый тест надо программировать вручную в объектных кодах), несколько субъективен (зависит от опыта программиста) и сильно зависит от типа программируемой тестовой задачи. При этом выбор теста имеет большое значение, так как всегда можно составить такую программу, на которой система оптимизации будет давать заведомо большой выигрыш, и наоборот, можно составить такую программу, на которой система оптимизации не дает никакого выигрыша.

Естественным методом сравнения эффективности различных систем оптимизации является трансляция одного и того же теста и затем сравнение полученных результатов. Разделяют четыре подхода к выбору тестовых задач, которые не исключают, а дополняют друг друга [34, 98]:

1) измерения на тестах, написанных специально для проверки качества реализации отдельных конструкций языка (специализированные тесты);

2) измерения на синтетических тестах, которые на основании профиля вычислительного центра, а именно статистических данных о его задачах, в определенных пропорциях содержат различные конструкции языка;

3) измерения на достаточно широком и разнообразном наборе классических задач (рассматривая свою задачу в заданном спектре классических задач, пользователь может оценить преимущества того или иного транслятора);

4) регистрация данных о реальных задачах в момент перехода пользователя с одной системы на другую.

На практике в исследованиях обычно применяют в той или иной мере все четыре подхода. Наиболее полно проведено сравнение качества трансляции по времени выполнения программ и объему занимаемой ими памяти для АЛЬФА-транслятора, трансляторов ТА-1 и ТА-2 [2]. Анализ и сравнение результатов трансляции тестов проведен и для трансляторов FOREX, ФОРТРАН-Дубна версия 16.7.73, ФОРТРАН-ГДР версия 20.9.74, АЛГОЛ-ГДР версия 6.6.75, ФОРШАГ версия 7.11.74 [98].

Для сравнения скорости выполнения объектной программы для разных трансляторов на разных ЭВМ [18] и качества

реализации языковых конструкций в этих трансляторах используется методика, предложенная в работе [91].

Непосредственное отношение к оценке эффективности оптимизационных процедур имеют метрологические исследования программ, выполняемых как в динамике, так в и статике. На основе анализа набора операций, используемых при решении задач на малогабаритной ЭВМ «Промінь», и частотных функций распределения операций определена методика увеличения быстродействия этого типа ЭВМ [52]. Получены статистические данные о составе операций в четырех различных инженерных и научных программах, разработанных для универсальной ЭВМ М-20 [99]. Получены статистические данные состава операций в программах четырех управляющих ЭВМ, используемых в сложных системах автоматического управления объектами. Характерной чертой исследуемых программ являются их высокая разветвленность и преобладание логических операций над арифметическими [71]. Получены также некоторые данные о частоте вхождения различных языковых конструкций в программы, типах структурного управления и глубины вложенности конструкций [20]. Помимо более точной оценки эффективности оптимизационных процедур, метрологические исследования позволяют: определять критические области программы с целью ее оптимизации, осуществлять настройку системы оптимизации на конкретные программы, выбирать эффективные алгоритмы оптимизации.

2. УРОВНИ ЯЗЫКОВ, НА КОТОРЫХ ВЫПОЛНЯЮТСЯ ОПТИМИЗАЦИОННЫЕ ПРЕОБРАЗОВАНИЯ

Если в системах оптимизации программ процесс выполнения оптимизационных преобразований функционально отделяется от процесса трансляции программы с одного языка на другой, т. е. эти два процесса выполняются последовательно и независимо друг от друга, то можно говорить об уровне языка, на котором выполняются оптимизационные преобразования, т. е. происходит собственно оптимизация программы.

Понятие уровня языка не формализуемо, но оно содержательно включает в себя параметры языка, характеризующие его сложность: разнообразие синтаксических конструкций; наличие различных форматов; множество типов операторов и т. д. Существенным свойством уровня языка является также то, что при переводе программы с языка одного уровня на другой язык более низкого уровня, в общем случае теряется часть информации, содержащейся в исходной программе.

Языки высокого уровня. Под языками высокого уровня обычно понимаются процедурно-ориентированные языки, такие как ПЛ/1, АЛГОЛ-60, ФОРТРАН и т. д. В настоящее время языки

высокого уровня, как правило, являются входными языками систем программирования и термин «оптимизация программ на уровне входного языка» в этом случае означает оптимизацию программ на уровне процедурно-ориентированного языка. Рассматриваемые языки свободны от таких ограничений (с точки зрения оптимизации программ), как знание устройств ЭВМ, ее машинных характеристик и т. д. Поэтому на этом уровне выполняются оптимизационные преобразования, инвариантные по отношению к техническим характеристикам ЭВМ, например к числу общих регистров, тому или иному набору команд. Однако такая инвариантность не означает полной независимости процесса оптимизации программ от характеристик ЭВМ. Если разница между машинно-независимыми и машинно-зависимыми языками достаточно ощутима, то разница между машинно-независимыми и машинно-зависимыми преобразованиями программ намного меньше, чем принято думать. Преобразование может быть машинно-независимым в том смысле, что оно не отражает машинно-зависимых свойств, но причины, по которым оно выполняется, могут определяться системой команд и ресурсами ЭВМ. Вследствие этого набор и последовательность выполнения таких преобразований на уровне входного языка зависят, хотя и неявно, от архитектуры ЭВМ. Например, система команд ЭВМ, содержащая богатый набор логических операций, расширяет возможности оптимизационных процедур, преобразующих логические выражения; разница во времени выполнения различных типов команд обусловливает целесообразность применения процедур, связанных с заменой в программах, написанных на языках высокого уровня, одних операций другими; наличие достаточного числа быстрых регистров расширяет возможности оптимизации циклов и т. д.

На уровне входного языка выполняются в основном оптимизационные процедуры, связанные с изменением структуры программы, перемещением и удалением операторов [3, 5, 116].

Разработка алгоритмов оптимизационных процедур, осуществляющих преобразования на уровне процедурно-ориентированных языков, в некотором смысле несколько проще разработки алгоритмов аналогичных процедур на уровне промежуточных языков и языков ассемблера по следующим причинам: во-первых, может быть использована та часть информации, которая содержится в тексте исходной программы, написанной на входном языке, но теряется при трансляции; во-вторых, конструкции программ, записанных на процедурно-ориентированных языках, достаточно крупны, что позволяет выполнять целостный анализ участков программ и последующие их преобразования.

Успешное решение задачи оптимизации программ на уровне входного языка особенно актуально для систем реального времени [77] и для некоторых специальных случаев автоматического

порождения программ на процедурно-ориентированных языках (например, сборкой программы из отдельных модулей, выводом программы из спецификаций или конкретизацией универсальной программы). Кроме того, успешное решение этой задачи приобретает особую актуальность в связи с развитием внутренних языков ЭВМ и появлением у ЭВМ четвертого поколения специализированных процессоров, предназначенных для выполнения программ, записанных на языках высокого уровня без предварительной их трансляции в машинные коды.

Промежуточные языки. Под промежуточным понимается язык уровня более низкого, чем входной, но более высокого, чем язык ассемблера. Процессу оптимизации программ на уровне промежуточного языка предшествует процесс ее трансляции с входного (процедурно-ориентированного) языка. Как правило, промежуточные языки являются машинно-независимыми для некоторого класса ЭВМ. Степень машинной независимости для разных классов ЭВМ может быть различной. Под машинной независимостью здесь понимается независимость от архитектуры ЭВМ в смысле, определенном в предыдущем подразделе. Выбор промежуточного языка определяется набором оптимизационных процедур, входным языком и, быть может, в какой-то степени внутренней структурой ЭВМ. Основным требованием, предъявляемым к промежуточному языку, является адекватность отображения программ, записанных на входном языке, и эффективность реализации оптимизационных процедур. Примером промежуточного языка может служить внутренний язык системы БЕТА [43].

Внутреннему языку (ВЯ) системы БЕТА ее разработчики уделяют особое внимание, так как он является фундаментальной концепцией обсуждаемой многоязыковой системы программирования [43]. При разработке ВЯ разработчики системы поставили две основные задачи: 1) добиться универсальности ВЯ, т. е. достаточно полного отображения на ВЯ программ, составленных на входных языках системы; 2) обеспечить эффективность и удобство выполнения оптимизационных преобразований на уровне ВЯ.

Большинство зарубежных систем оптимизации выполняют оптимизационные преобразования на уровне промежуточного языка, задаваемого в виде триад или тетрад [35]. Применение одной из этих форм определяется каждым конкретным случаем. Следует заметить, что каждая из этих форм обладает определенными недостатками. Например, для тетрад требуется описание временной переменной, а триады сложны в обработке.

На уровне промежуточного языка могут выполняться такие же оптимизационные процедуры, как и на уровне входного языка. Однако алгоритмы этих оптимизационных процедур, области оптимизации и оптимизационный эффект существенно отличаются. Другой набор оптимизационных преобразований, выпол-

няемых на уровне промежуточного языка, связан с распределением памяти как на линейных участках, так и в пределах всей программы.

Языки ассемблера. Оптимизация на уровне языков ассемблера связана в основном с экономией памяти, оптимизацией использования различных уровней памяти, в частности быстрых регистров, и с заменой одних групп команд на другие, более эффективные [35]. Алгоритмы оптимизационных процедур, выполняемых на уровне языков ассемблера, как правило, очень существенно зависят от структуры ЭВМ, исключение составляют лишь некоторые алгоритмы экономии памяти [65]. Оптимизация использования регистров заключается, с одной стороны, в максимальном использовании быстрых регистров, а с другой (если потребность в регистрах превосходит их число),— в оптимизации использования этих регистров. Алгоритмы замены одних групп команд другими, более эффективными, практически полностью связаны с конкретной ЭВМ.

Взаимосвязь различных уровней оптимизации. В общем случае процесс оптимизации программ описывается многоступенчатой схемой. Первоначально осуществляется оптимизация программ на уровне входного процедурно-ориентированного языка, где выполняются оптимизационные преобразования, инвариантные по отношению к ЭВМ. Затем преобразованная оптимизационными процедурами программа транслируется на некоторый промежуточный язык, на уровне которого выполняется ряд специфических оптимизационных преобразований, и т. д. Заметим, что промежуточных языков может быть несколько. На последней ступени оптимизации, выполняемой на уровне языка ассебмлера, учитываются все особенности ЭВМ. Как правило, при таком подходе достаточно трех ступеней оптимизации для языков: процедурно-ориентированного; промежуточного; ассемблера.

Повышение уровня входного языка, естественно, приводит к увеличению числа ступеней оптимизации программ. Например, оптимизация может состоять из четырех ступеней [106] для языков: очень высокого уровня; подобного Паскалю; уровня BLISS [194]; ассемблера.

Все ступени оптимизации программ дополняют друг друга и являются частями общего процесса оптимизации. Для каждой ступени оптимизации имеется набор оптимизационных процедур, наиболее эффективно выполняемых на языке соответствующего уровня. Например выполнение процедуры оптимизации выражений наиболее эффективно на уровне входного языка, в то время как выполнение оптимизации использования быстрых регистров наиболее эффективно на уровне языка ассемблера.

Чем выше уровень языка, на котором осуществляется оптимизация, тем значительней изменения программы и алгоритма решения задачи, осуществляемые при выполнении каждого опти-

мизационного преобразования. Оптимизационный эффект, получаемый в результате изменения алгоритма решения задач с целью оптимизации, как правило, превышает оптимизационный эффект, получаемый в результате преобразования программы без изменения алгоритма решения задачи. Иначе говоря, хороший алгоритм, плохо реализованный, как правило, эффективней плохого алгоритма, хорошо реализованного. В качестве одного из подходов к изменению алгоритма решения задачи можно привести оптимизацию программ на основе метода формализованных технических заданий [69]. Таким образом, чем выше уровень языка, тем больше оптимизационный эффект, получаемый в результате выполнения некоторой оптимизационной процедуры. Однако вместе с тем это никак не умаляет значения оптимизации программ на языках более низкого уровня, так как чем ниже уровень языка, тем больше существует возможностей оптимизации программ на этом уровне. Другими словами, чем выше уровень языка, на котором написана программа, тем меньше вероятность ее улучшить, однако если эту программу можно улучшить, то это будет существенное улучшение. И чем ниже уровень языка, на котором написана программа, тем больше вероятность ее улучшить, однако каждое такое улучшение лишь немного повысит качество программы в целом. Такой баланс между возможностью улучшения программы и степенью улучшения не позволяет пренебрегать оптимизацией на языках любого уровня.

Языки сверхвысокого уровня. Под языками сверхвысокого уровня понимаются языки проектирования программ. Рассмотренные выше языки всех уровней являются языками программирования или, иначе говоря, языками кодирования программ. Языки проектирования предназначены для разработки алгоритмов и тесно связаны с методами проектирования программ. Например, можно привести метод формализованных технических заданий [28]; структурный подход к программированию [92]; метод последовательного уточнения программы [118]. В настоящее время языки сверхвысокого уровня не являются достаточно развитыми, чтобы можно было делать достаточно глубокие обобщения. Однако, тем не менее, уже получены результаты, позволяющие говорить о существовании возможности решения задачи оптимизации программ и на этом уровне языков. В частности, в рамках метода формализованных технических заданий, выполняя эквивалентные преобразования функциональных моделей, содержащих некоторый алгоритм решения задачи, можно добиться улучшения создаваемого алгоритма [25, 66]. Можно также утверждать, что для языков сверхвысокого уровня справедливо правило, согласно которому при преобразовании программ (алгоритмов), записанных на этих языках, можно получить значительный оптимизационный эффект, однако вероятность выполнимости таких преобразований невелика.

3. ОБЛАСТИ ОПТИМИЗАЦИИ

Под областью оптимизации понимается часть программы, т. е. множество операторов, над которыми выполняются оптимизационные преобразования независимо от других операторов программы, т. е. независимо от контекста. В частности, областью оптимизации может быть вся программа.

Представление программы в виде графа. При изучении свойств программ с целью выполнения над ними оптимизационных преобразований естественным является рассмотрение не самих программ, а их моделей, сохраняющих рассматриваемые свойства программ, и в то же время независимых от языка, на котором они составлены. В качестве таких моделей будем использовать представление программ в виде ориентированных графов (ниже будем употреблять просто термин граф).

Напомним некоторые определения теории графов [16]. Рассмотрим два множества $X = \{x\}$ и $Y = \{y\}$. Обозначим через Γ некоторое отображение X в Y, т. е. $y \in \Gamma x$. Отображение Γ в общем случае может быть неоднозначным. Пусть $A \subset X$, тогда образом множества A называется множество

$$\Gamma A = \bigcup_{x \in A} \Gamma x.$$

Отображение Γ^{-1}, обратное отображению Γ, определяется следующим образом:

$$\Gamma^{-1} y = \{x \mid y \in \Gamma x\}.$$

Пусть $B \subset Y$, тогда

$$\Gamma^{-1} B = \{x \mid \Gamma x \cap B \neq \varnothing\}.$$

Под графом G понимается пара (X, Γ), где X — некоторое множество вершин графа, а Γ — отображение X в X. Граф будем обозначать символом $G = (X, \Gamma)$. Пара вершин (x, y), в которой $y \in \Gamma x$, называется дугой графа. Дуги будем обозначать $u = (x, y)$. Вершина x называется началом дуги $u = (x, y)$ (начальной вершиной), а вершина y — концом (конечной вершиной). Будем говорить, что дуга u входит в вершину y и выходит из вершины x.

Путем в графе $G = (X, \Gamma)$ называется такая последовательность дуг $(u_1, u_2, \ldots)$, что конец каждой предыдущей дуги совпадает с началом следующей. Путь, в котором никакая вершина не встречается дважды, называется простым.

Контур — это конечный путь $(u_1, u_2, \ldots, u_n)$, у которого начальная вершина дуги u_1 совпадает с конечной вершиной дуги u_n.

Подграфом графа (X, Γ) называется граф вида (A, Γ_A), где $A \subset X$, а отображение Γ_A определено следующим образом:

$$\Gamma_A x = \Gamma x \cap A.$$

Рассмотрим некоторый подграф R графа G. Будем говорить, что подграф R является одновходовым, если существует только одна вершина (входная) $x_н \in R$, в которую входят дуги, выходящие из вершин, не принадлежащих R. Аналогично будем говорить, что подграф R является одновыходовым, если существует только одна вершина (выходная) $x_к \in R$, из которой выходят дуги, входящие в вершины, не принадлежащие подграфу R. (Здесь приведено несколько основных определений, остальные, необходимые для изложения поня-

тий теории графов, вводятся в соответствующих местах, при этом соблюдается терминология, принятая в работе [16]).

Каждой программе можно поставить в соответствие два графа, определяемых двумя типами связей, существующими в программе: управляющими (логическими) и информационными. Будем говорить, что

между операторами Q и S существует управляющая связь, если оператор S может выполняться непосредственно после оператора Q, т. е. оператор Q (быть может при выполнении некоторых условий) передает управление на оператор S;

между операторами Q и S существует информационная связь, если оператор S использует значения некоторых переменных, определяемых оператором Q.

Под управляющим (логическим) графом понимается такой граф, вершины которого соответствуют операторам, а дуги — управляющим связям между операторами.

Под информационным графом понимается такой граф, вершины которого соответствуют операторам, а дуги — информационным связям между операторами.

Например, участку программы

```
1   X1 = A + B
2   X2 = X1 + C
3   IF (X1. GE. X2) GOTO 33
4   X1 = X2 + B
5   GOTO 66
633 X3 = X1 + D
766 X4 = X1 + X3
```

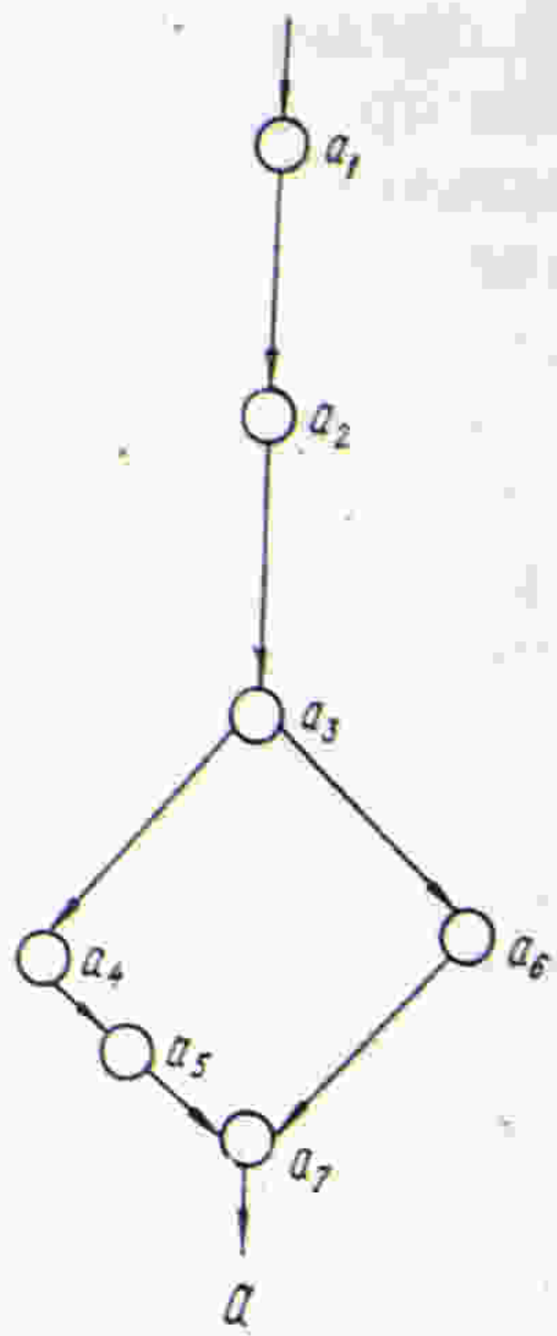
соответствуют управляющий (рис. 1, *а*) и информационный (рис. 1, *б*) графы. Здесь и далее буквой *а* с индексом обозначены вершины графа, соответствующие операторам программы, и значение индекса является порядковым номером оператора в программе или в рассматриваемом ее участке (порядковые номера находятся слева от операторов соответствующего участка программы). На информационном графе рядом с дугами указаны переменные, значения которых определяются и используются операторами, являющимися начальной и конечной вершиной соответствующей дуги.

В дальнейшем под графом программы будет по умолчанию подразумеваться управляющий граф, а при рассмотрении информационного графа будут делаться специальные оговорки. Кроме того, для удобства изложения материала в некоторых случаях понятие вершины графа, соответствующей оператору программы, будет заменяться, без специальных оговорок, понятием самого оператора программы и наоборот. При этом все определения, свя-

занные с графами программ, переносятся соответствующим образом на программы и наоборот.

Различают следующие структуры в программах, являющиеся областями оптимизации: оператор; линейный участок (блок); зона; интервал; гамак; фрагмент; программа.

Оператор. В общем случае оператору соответствует одна вершина графа программы. В пределах одного оператора, как правило, выполняются оптимизационные процедуры, обеспечивающие преобразование одних типов данных в другие, а также замену одних операций другими, более эффективными, например,



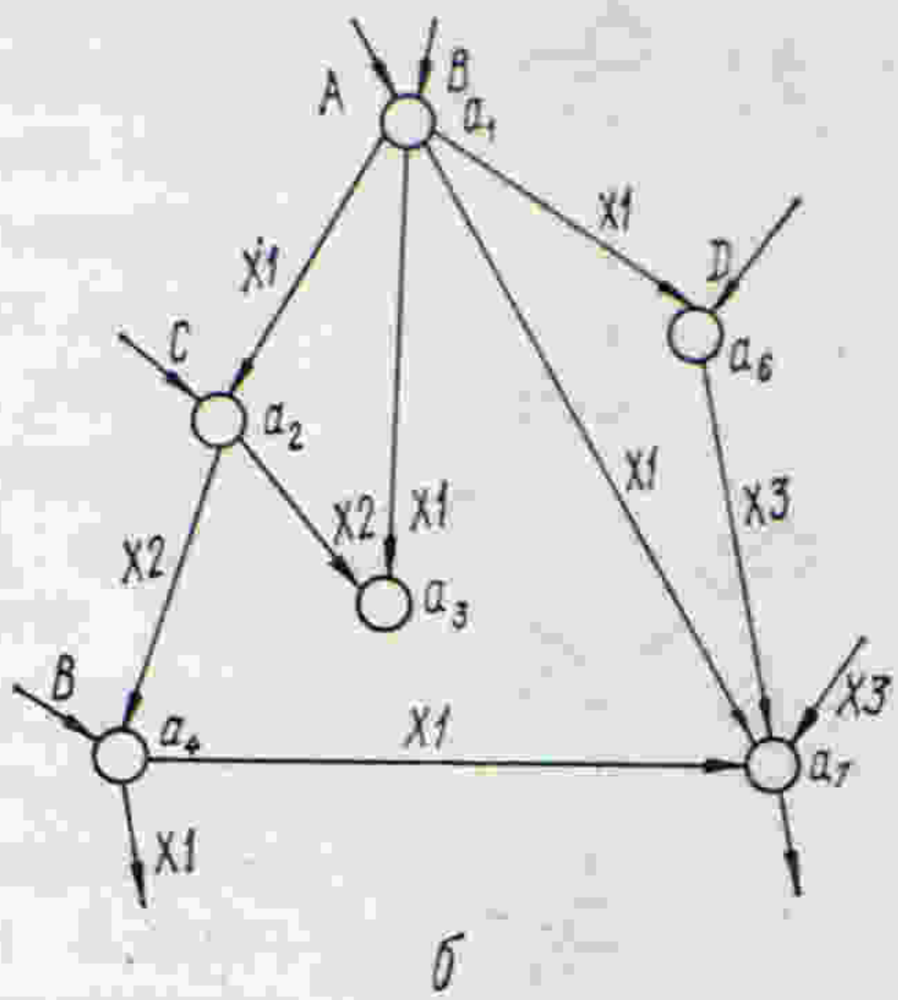


Рис. 1

Рис. 2

умножение сложением, деление умножением и т. д. Выполняется также оптимизация арифметических выражений [11]. Несмотря на кажущуюся простоту рассматриваемой области оптимизации, следует заметить, что даже здесь при выполнении некоторых оптимизационных преобразований возникают трудности. Так, например, эффективный алгоритм полной оптимизации арифметических выражений в пределах одного оператора в настоящее время не известен. Однако разработан ряд эвристических методов, позволяющих получать хорошие практические результаты. Достаточно полная теория оптимизации создана только для выражений, не содержащих одинаковых операндов [14].

Линейный участок. Под линейным участком понимается одновходовый подграф, не содержащий контуров, такой, что простой путь, исходящий из входной вершины, проходит через все остальные вершины этого подграфа.

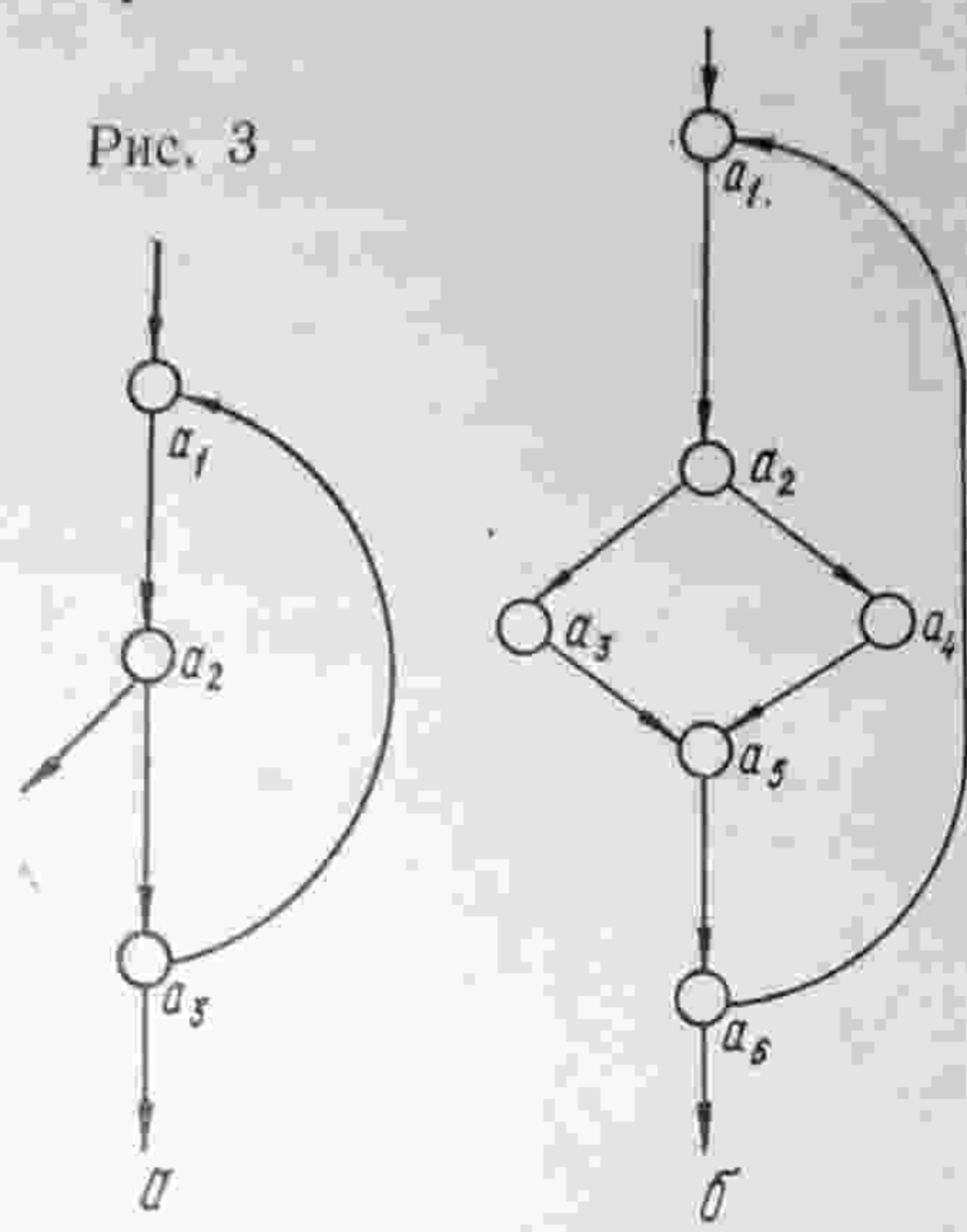
Программа, соответствующая линейному участку, является последовательностью операторов, перед которыми (за исключением первого) не должно быть меток. Действительно, метка либо является входом, а вход у линейного участка только один (пер-

вый оператор), либо эта метка содержательно излишняя и ее можно удалить. На рис. 2, например, показан граф, соответствующий следующему линейному участку программы:

```
1  X1 = A + B
2  X2 = C*D - 5
3  X3 = X1/X2
4  X1 = X2 + B
```

Алгоритмы реализации процедур, областью оптимизации которых является линейный участок, существенно проще алгоритмов реализации аналогичных процедур, областью оптимизации которых является, например, зона или гамак, так как анализ управляющих и информационных связей линейного участка не вызывает трудностей. Несмотря на кажущуюся простоту линейных участков, их оптимизации уделяется существенное внимание [14, 35]. Это связано с тем, что любые более сложные структуры в конечном счете состоят из линейных участков и эффективность линейных участков в какой-то степени характеризует эффективность всей структуры. На линейных участках, как правило, выполняются оптимизационные процедуры, связанные с экономией памяти [97] и рациональным проведением арифметических вычислений. Кроме того, в языках ассемблера на линейных участках осуществляется замена одной группы машинных команд другой, которая в конкретном контексте работает более эффективно.

Рис. 3



Зона. Зоной называется сильно связанный подграф, т. е. подграф, у которого существуют пути между двумя вершинами (рис. 3, *а*). Одновходовые и одновыходовые зоны, как правило, называют циклами (рис. 3, *б*). Большинство языков высокого уровня имеют специальные средства для описания циклов, что легко позволяет осуществлять их поиск в программе на этапе синтаксического анализа.

Характерной особенностью зон является то, что небольшое число операторов, составляющих зону, может потреблять значительную часть времени выполнения программы за счет многократного выполнения этих операторов. Поэтому зоны являются наиболее важными областями оптимизации. Методы оптимизации зон и их простейшего варианта — циклов — начали разрабатываться практически одновременно с постановкой задачи оптимизации

программ и в настоящее время имеется ряд хорошо зарекомендовавших себя алгоритмов и значительный опыт в реализации этих алгоритмов [81]. К наиболее распространенным процедурам, областью оптимизации которых является зона или цикл, можно отнести: чистку зон, объединение циклов, развертку циклов и преобразование рекурсивно-определяемых переменных [7].

Интервал. Рассмотрим определение, используя которое можно любой граф единственным образом разбить на интервалы [14]. Интервалом $G(a)$ с головной вершиной a называется такой подграф графа G, что

1) $a \in G(a)$;

2) если вершина a_x, еще не включенная в $G(a)$, не является входной, и все дуги, входящие в a_x, выходят из вершин, принадлежащих $G(a)$, то a_x включается в $G(a)$;

3) повторяется шаг (2) до тех пор, пока не останется вершин, которые можно добавить к $G(a)$.

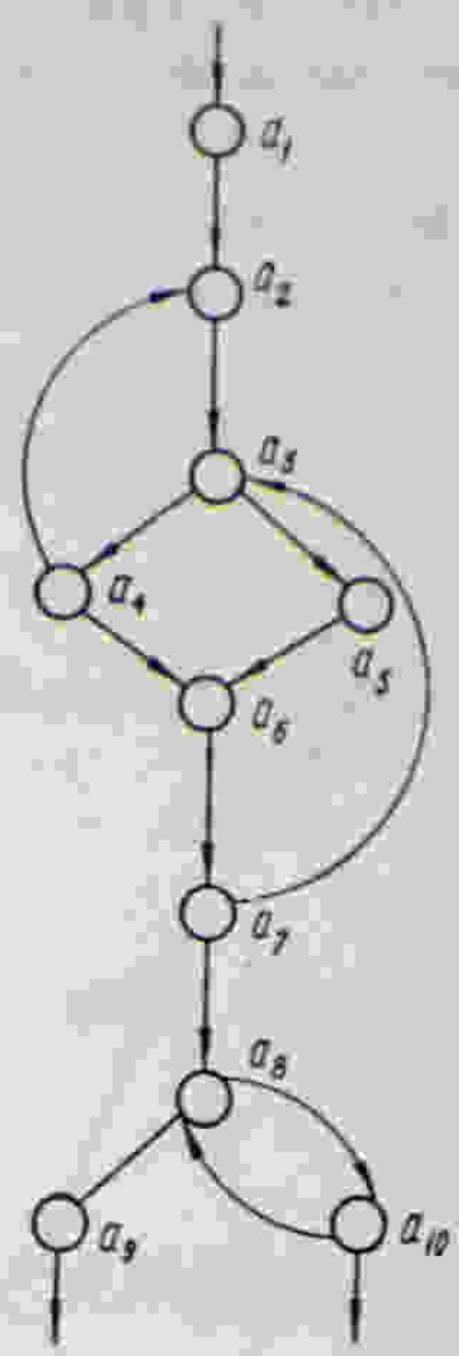


Рис. 4

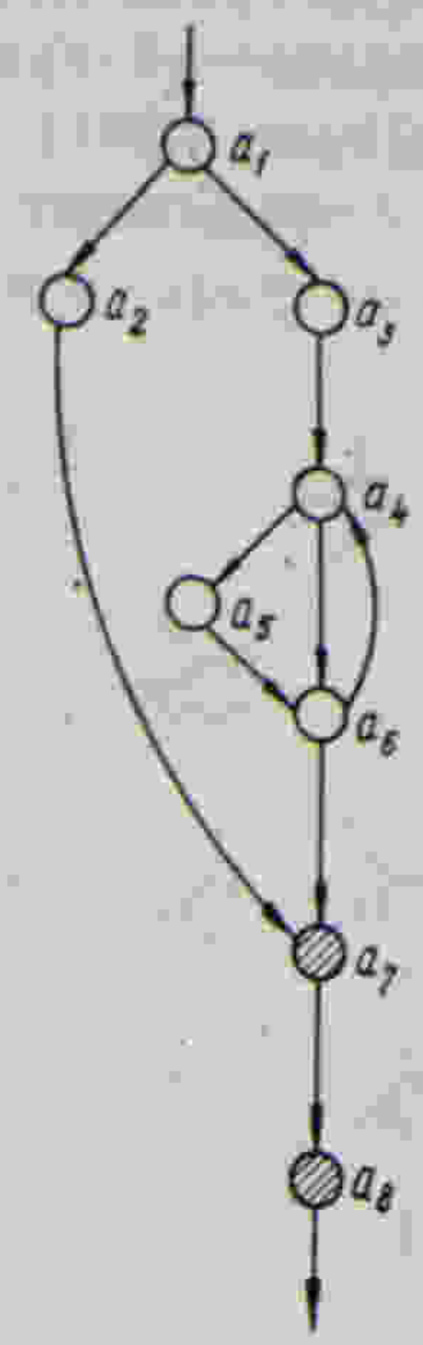


Рис. 5

На графе, показанном на рис. 4, интервалами являются следующие подграфы:

$$G(a_1) = \{a_1\};$$
$$G(a_2) = \{a_2\};$$
$$G(a_3) = \{a_3,\ a_4,\ a_5,\ a_6,\ a_7\};$$
$$G(a_8) = \{a_8,\ a_9,\ a_{10}\}.$$

Разбиение графа программы на интервалы позволяет осуществлять факторизацию [6], т. е. преобразование, при котором интервалы исходного графа рассматриваются как вершины результирующего графа, являющегося обобщением исходного. Используется оно, как правило, на этапе анализа программ. В результате такого анализа собирается информация, необходимая для выполнения ряда оптимизационных процедур.

Гамак. Подграф $G' = (A, \Gamma_A)$ графа G называется гамаком [54, 84], если

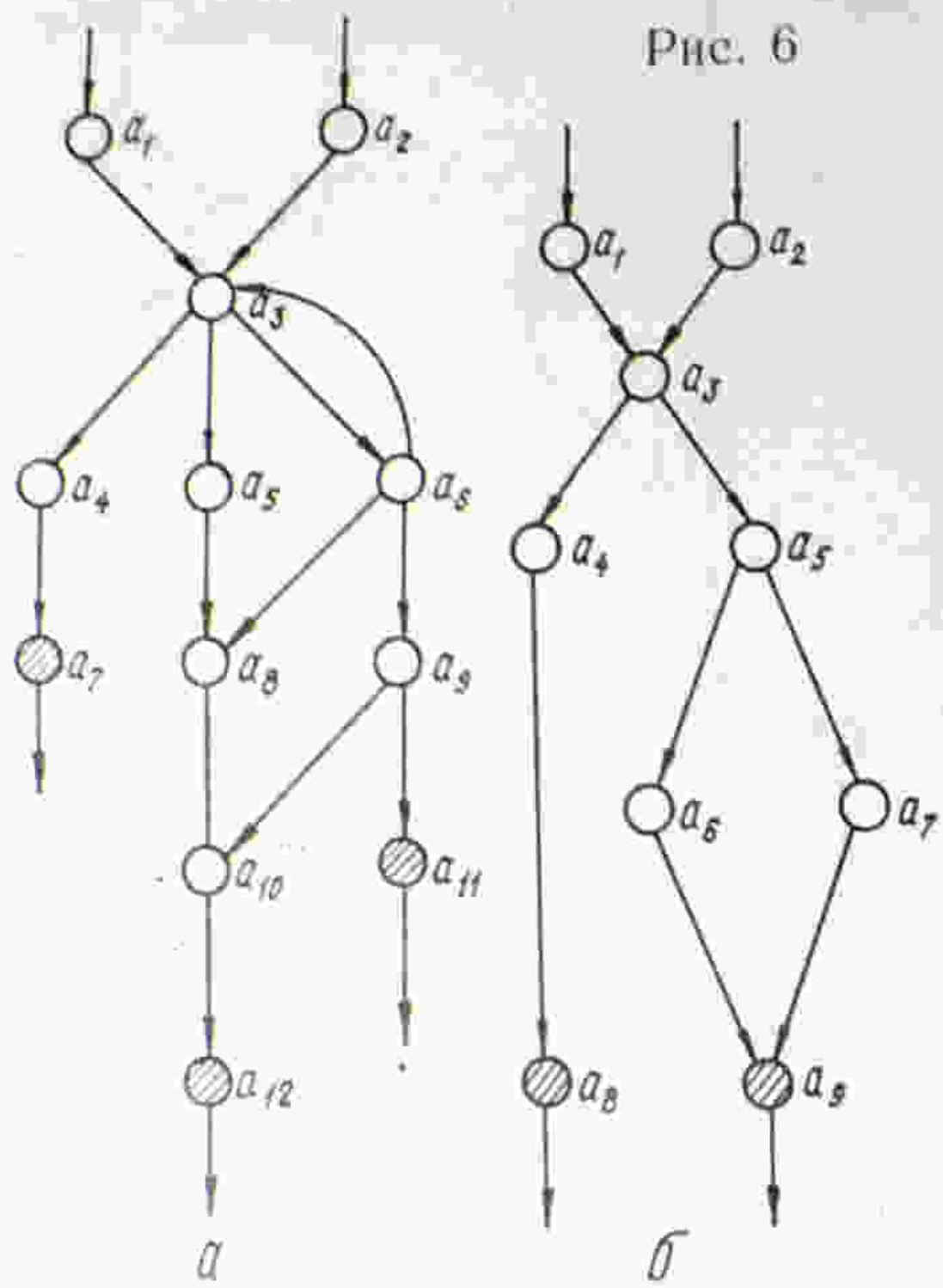
имеется единственная вершина $x_н \in A$, такая, что либо $\Gamma Y \cap A = x_н$, где $Y = X/A$, либо $x_н = x_0$, где x_0 — входная вершина графа;

имеется единственная вершина $x_к \in Y$, такая, что $\Gamma A \cap Y = x_к$ и $\Gamma^{-1} x_к \cap Y = \varnothing$.

На графе, показанном на рис. 5, подграф, включающий незаштрихованные вершины, является гамаком. Если зоны можно рассматривать как обобщение циклов, то гамаки можно рассматривать как обобщение операторов с одним входом и одним выходом, например операторов присваивания. К наиболее распространенным процедурам, областью оптимизации которых являются гамаки, можно отнести чистку гамаков и втягивание в гамак [81].

Фрагмент является обобщением гамака и представляет собой подграф, имеющий несколько входных и несколько выходных вершин, ему не принадлежащих. Проводя аналогию, можно сказать, что гамак является таким же частным случаем фрагмента, как цикл частным случаем зоны, т. е. гамак является одновходовым и одновыходовым фрагментом. Такая область оптимизации как гамак была выделена раньше чем фрагмент. Введение в рассмотрение фрагмента обусловлено стремлением увеличить оптимизационный эффект процедур путем расширения области их действия. Несмотря на то что фрагмент является обобщением гамака, его следует рассматривать как отдельную область оптимизации, так как число анализируемых управляющих и информационных связей программы существенно меняет алгоритмы реализации оптимизационных процедур. Заметим, что фрагмент является структурой, на управляющие и информационные связи которой практически не накладывается никаких ограничений, и поэтому к фрагментам применимы только самые общие методы оптимизации.

Рис. 6



На рис. 6 подграфы, включающие незаштрихованные вершины, являются фрагментами. К основным процедурам, областью оптимизации которых является фрагмент, можно отнести чистку фрагментов. Содержательно эта процедура аналогична процедуре чистки гамаков, однако наличие нескольких входов и выходов у фрагмента усложняет алгоритм реализации этой процедуры и накладывает дополнительные ограничения на ее выполнение.

Программа — это максимально возможная область оптимизации. Алгоритмы оптимизационных процедур, ориентированные на область оптимизации, являющуюся всей программой, наиболее сложны и трудно реализуемы, так как в этом случае не нак-

ладывается никаких ограничений на управляющие и информационные связи области оптимизации. Однако вместе с тем выполнение таких процедур позволяет получить максимальный оптимизационный эффект.

Взаимосвязь областей оптимизации. Конкретные структуры рассмотренных областей: зон, интервалов, гамаков и фрагментов могут быть самыми разными, и при этом они могут быть связаны между собой. В частности, зоны могут содержать гамаки (операторы a_2, a_3, a_4 на рис. 3, *б* составляют гамак) и наоборот, гамаки

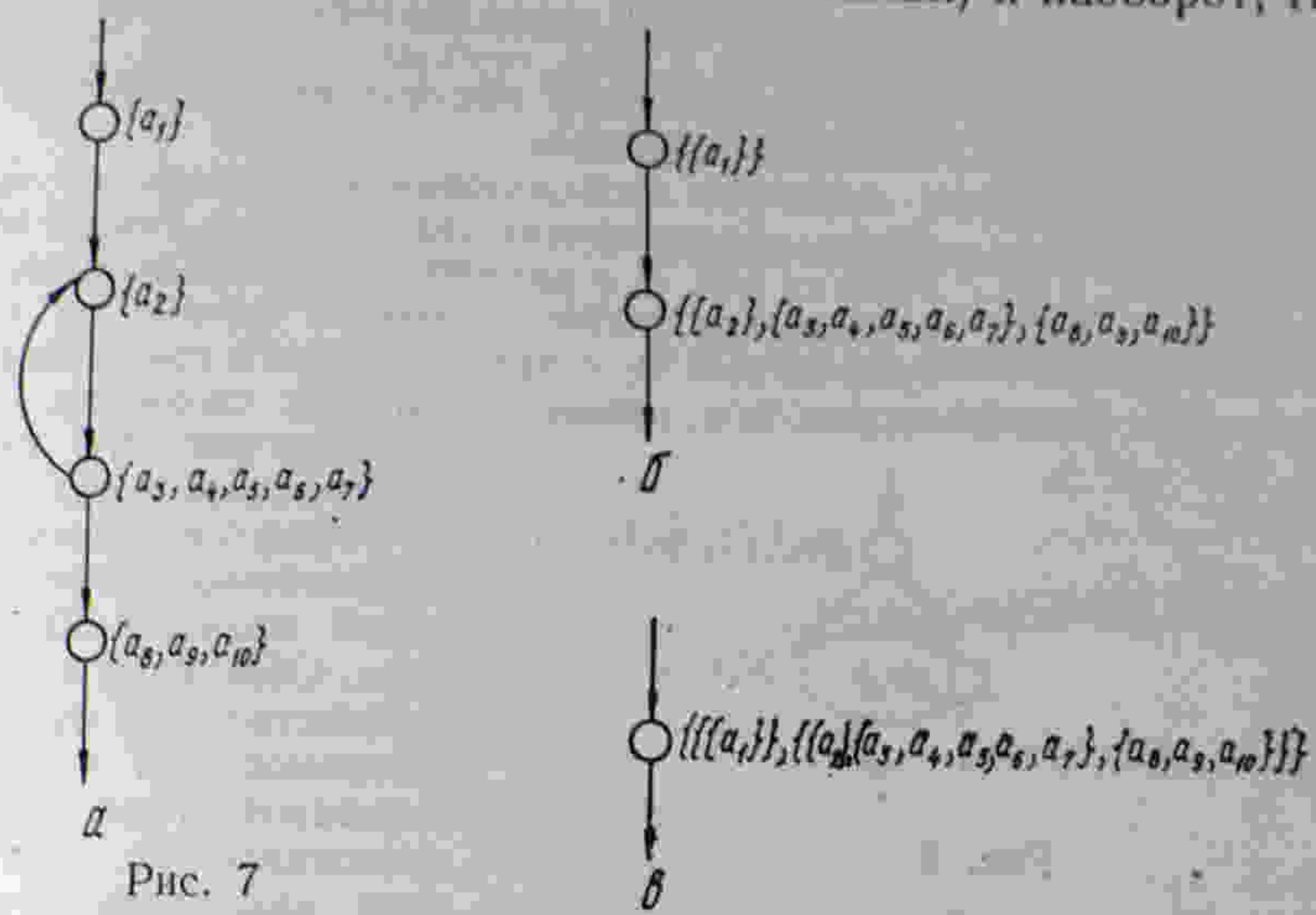


Рис. 7

могут содержать зоны (операторы a_4, a_5, a_6 и a_4, a_6 на рис. 5 составляют зону). Фрагменты могут содержать гамаки (операторы a_5, a_6, a_7 на рис. 6, *б* составляют гамак), зоны (операторы a_3, a_6 на рис. 6, *а* составляют зону) и т. д.

В соответствии с различными стратегиями оптимизации граф программы разбивается на структуры различного типа. Рассмотрим взаимосвязи между такими разбиениями, для чего введем понятия сводимого, зонно-одновходового и аранжируемого графов.

Рассмотрим некоторый граф G и разобьем его на интервалы. Затем, приняв интервал за элементарную единицу, произведем факторизацию [6]. Граф G_1, полученный в результате факторизации, назовем производным графом от G. Аналогично граф G_1 можно разбить на интервалы, выполнить факторизацию и получить граф G_2, производный от графа G_1. Нетрудно видеть, что если в некотором графе существует интервал, содержащий более одной вершины, то при переходе к производному графу число вершин в нем будет убывать. В результате такого процесса будет получен граф G_n, от которого нельзя получить производный граф. Граф G_n называется предельным (пределом) для графа G. Если G_n состоит из одной вершины, то граф G называется сводимым. Граф программы можно единственным образом разбить на интервалы [14]. Любой несводимый граф можно преобразовать в *сводимый* граф. Суть такого преобразования заключается в замене некоторых вершин графа, в которые входит более одной дуги на несколько одинаковых копий этих вершин, по одной на каждую дугу.

Для большинства реальных программ их управляющие графы являются сводимыми.

Граф, производный от графа, показанного на рис. 4, содержит следующие интервалы (рис. 7, *а*):

$$G(\{a_1\}) = \{\{a_1\}\};$$

$$G(\{a_2\}) = \{\{a_2\},\ \{a_3, a_4, a_5, a_6, a_7\},\ \{a_8, a_9, a_{10}\}\}.$$

Граф, производный от графа, показанного на рис. 7, *а*, содержит один интервал (рис. 7, *б*):

$$G(\{\{a_1\}\}) = \{\{\{a_1\}\},\ \{\{a_2\},\ \{a_3, a_4, a_5, a_6, a_7\},\ \{a_8, a_9, a_{10}\}\}\},$$

и на следующем этапе факторизации этот граф преобразуется в граф, показанный на рис. 7, *в*. Результаты факторизации показывают, что граф на рис. 4 является сводимым.

На рис. 8, *б* изображен предельный граф от графа, приведенного на рис. 8, *а*. Таким образом, граф на рис. 8, *а* является несводимым. В результате выполнения операции замены вершин их копиями граф на рис. 8, *б* преобразуется в граф, показанный на рис. 9, *а*, который является сводимым, как это видно из рис. 9, *б*, где показан его предельный граф.

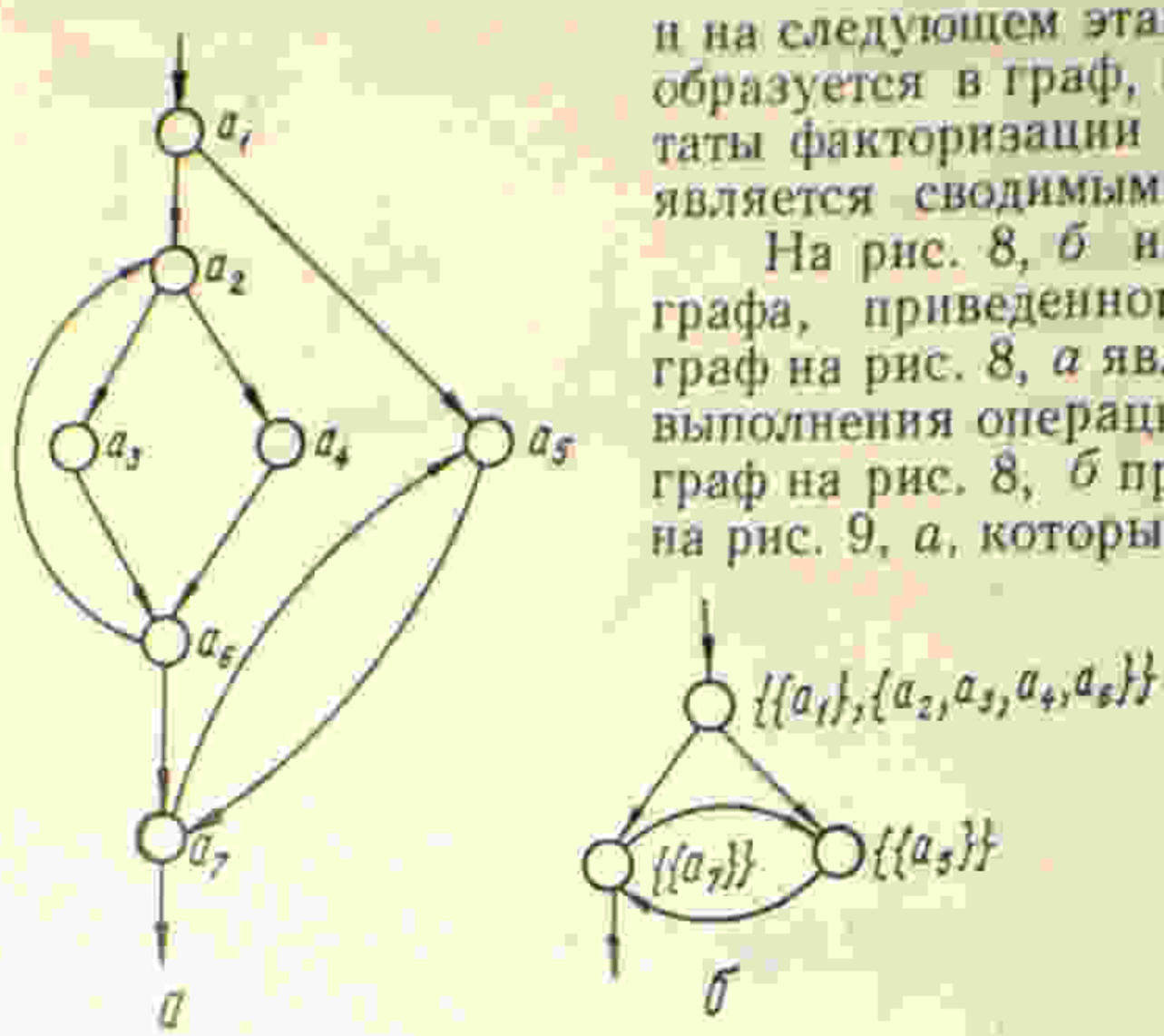


Рис. 8

Граф G называется *зонно-одновходовым*, если одновходовой является каждая его зона. Обозначим через $R(G)$ множество всех путей графа, начинающихся его входной вершиной. Граф называется *аранжируемым*, если существует аранжировка его вершин, т. е. такое линейное упорядочение (нумерация), в котором конечная вершина (ее номер) любого простого пути $R \in R(G)$ строго больше любой другой вершины (ее номера) этого пути, и нумерация вершин плотная [84].

Понятия сводимых, зонно-одновходовых и аранжируемых графов используются при выделении различных областей оптимизации для реализации оптимизационных процедур на этапе анализа программ. Сводимый граф обладает свойством зонной одновходовости и аранжируемости [55]. При этом любой несводимый граф путем «расщепления» некоторых его вершин всегда можно преобразовать в аранжируемый и в зонно-одновходовый граф.

Структура и размеры областей оптимизации существенно влияют на оптимизационный эффект выполняемых процедур. Например, оптимизационный эффект процедуры оптимизации выражений [11], у которой областью оптимизации является вся программа, в общем случае существенно превосходит оптимизационный эффект той же процедуры, но у которой областью оптимизации является линейный участок этой программы.

Локальная и глобальная оптимизации. Оптимизация программ в зависимости от вида области оптимизации подразделяется на глобальную и локальную. Под глобальной оптимизацией про-

грамм понимается такая оптимизация, при которой выполнение оптимизационных процедур требует учета управляющих и информационных связей на участке, превосходящем линейный участок, например в зоне или гамаке. Под локальной оптимизацией программ понимается оптимизация внутри линейного участка или отдельного оператора.

Следует отметить, что глобальная и локальная оптимизации программ являются взаимосвязанными процессами. Например, возможности локальных оптимизационных процедур увеличиваются за счет предварительного применения глобальных оптимизационных процедур, и, наоборот, возможности глобальных оптимизационных процедур в некоторых случаях увеличиваются за счет предварительного выполнения локальных.

Локальная оптимизация, как правило, осуществляется за один про-

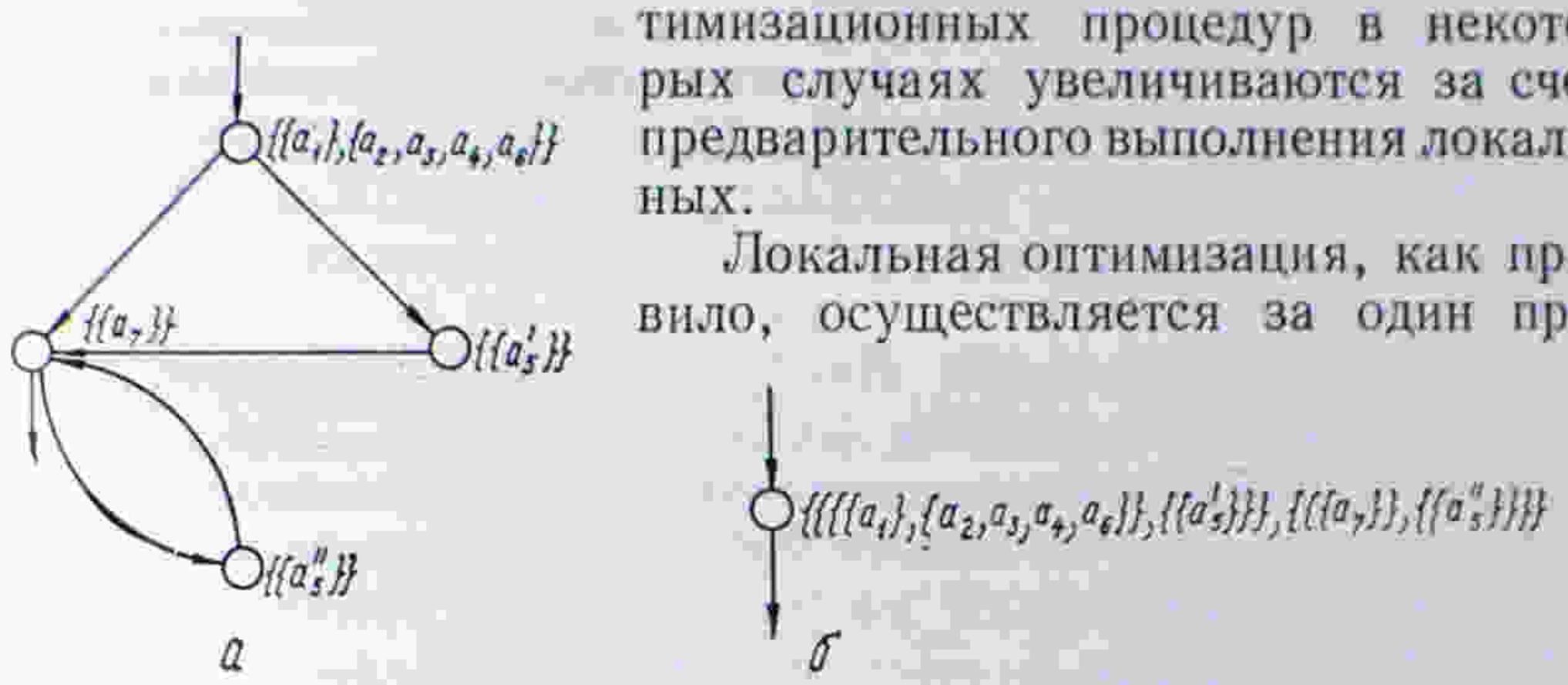


Рис. 9

смотр программы. Для большинства преобразований, выполняющих глобальную оптимизацию программ, требуется практически одинаковая информация о связях, существующих между переменными и операторами в программе. Процесс глобальной оптимизации состоит из двух этапов, требующих нескольких просмотров программы.

На первом этапе осуществляется анализ управляющего и информационного графов программы, отражающих два типа связей — управляющих и информационных, существующих между переменными и операторами в программе [45, 46, 60]. В процессе этого анализа могут выполняться следующие основные операции:

укладка операторов в виде линейной последовательности, что необходимо для выделения некоторых областей оптимизации и выполнения оптимизационных процедур;

выделение областей оптимизации программы (линейные участки, зоны, интервалы, гамаки, фрагменты), в терминах которых сформулированы основные оптимизационные преобразования;

получение некоторой информации о частоте выполнения отдельных участков программы, что имеет существенное значение для определения общей стратегии оптимизации.

На втором этапе на основе анализа управляющего и информационного графов программы осуществляется выполнение опти-

мизационных процедур. Для анализа и преобразования управляющих и информационных графов программы разработано несколько подходов.

На основе алгебры алгоритмов осуществляется преобразование микропрограмм, приводящее их к более эффективному виду [24, 25]. Существенным моментом, определяющим возможность таких преобразований, является запись микропрограммы в регулярной форме. Доказано существование алгоритма преобразования произвольных микропрограмм в регулярную форму и построен алгоритм, осуществляющий это преобразование [25]. Этот метод применяется для оптимизации по времени работы многих операторов как числового, так и не числового характера [66] и используется при исследованиях по полугрупповой эквивалентности дискретных преобразователей. Получены условия дискретного преобразователя наилучшего по абсолютному быстродействию относительно полугруппы с левым сокращением и неразложимой единицей [67].

Ряд оптимизационных преобразований основывается на использовании инвариантных соотношений [68]. Такие оптимизационные преобразования применяются при оптимизации программ путем введения и устранения избыточных вычислений [69]. Возможности использования рассматриваемых преобразований зависят от построения полных систем инвариантных соотношений. Такие системы построены для абсолютно свободной алгебры данных (алгебры термов), в которой нет нетривиальных тождественных зависимостей [61], и для программ, алгебрами данных которых являются свободные группы и полугруппы [63] и свободные абелевы группы и полугруппы [62]. Имеется ряд примеров применения инвариантных соотношений для оптимизации программ [51].

Для построения моделей программ, содержащих указатели, значениями которых являются имена других величин, используются схемы с косвенной адресацией (КА-схемы) [57]. Для построения моделей программ, не содержащих указатели, используются Р-схемы и О-схемы, которые при соблюдении некоторых условий можно преобразовывать друг в друга с сохранением эквивалентности [56].

ГЛАВА 2
ОПТИМИЗАЦИОННЫЕ ПРОЦЕДУРЫ

1. ОСОБЕННОСТИ ОПИСАНИЯ ОПТИМИЗАЦИОННЫХ ПРОЦЕДУР

Характер написания программы, ее особенности, в том числе неоптимальные конструкции, во многом зависят от квалификации программиста, уровня языка, а также от программируемого алгоритма. Широкий диапазон изменения этих факторов определяет то большое разнообразие неоптимальных конструкций, которые допускают в своих программах даже опытные программисты. Обобщение и анализ таких конструкций представляют собой трудную и интересную задачу, полное решение которой позволило бы, с одной стороны, более эффективно решить многие вопросы оптимизации программ, а с другой,— лучше понять процесс программирования с психологической точки зрения.

К настоящему времени сделан ряд попыток обобщить и классифицировать неоптимальные конструкции, которые встречаются в программах начинающих и более опытных программистов, и разработать соответствующие оптимизационные процедуры [4, 72, 81, 88, 104, 106].

Приводимый набор оптимизационных процедур не претендует на полноту, он содержит основные процедуры, которые находятся в обращении и хорошо осмыслены пользователями. При определении набора учитывалось то, что разнообразие практических задач оптимизации требует построения целого спектра различных по мощности и сложности множеств оптимизационных процедур. Это необходимо для того, чтобы в зависимости от особенностей рассматриваемого класса программ, а также проектных характеристик системы оптимизации можно было выбирать наиболее подходящее множество таких процедур и последовательность их применения.

Среди оптимизационных процедур имеются такие, выполнение которых не связано с процессом трансляции, а только с построением алгоритма программы: при выполнении таких процедур, как правило, удаляются некоторые операторы или операции, что всегда приводит к улучшению программы. Имеются также процедуры, эффект от выполнения которых основывается на некоторых особенностях процесса трансляции. Эти процедуры, как правило, связаны с заменой одних операций на другие. Для большинства трансляторов такие процедуры дают положительный оптимизационный эффект, однако могут встречаться трансляторы, для которых выполнение таких процедур приводит к отрицательному эффекту, т. е. к ухудшению качества программы. Поэтому перед применением процедур следует предварительно изучить особенности процесса трансляции или же провести эксперимент по выявлению процедур с отрицательным эффектом по отношению к данному транслятору. Более того, авторы не рекомендуют пользователям применять процедуры без предварительных экспериментов с используемым транслятором.

Критерием оптимизации описываемых процедур является время выполнения программы и/или объем занимаемой ею памяти. Однако, как уже отмечалось, оптимизация времени выполнения программы и оптимизация объема занимаемой ею памяти, в общем случае, находятся в некотором «противоречии» друг с другом. Поэтому среди описанных оптимизационных процедур есть такие, которые в этом смысле «противоречат» друг другу. Одни из них экономят память, но вызывают при этом проигрыш по времени, другие экономят время, но вызывают при этом проигрыш по памяти. Кроме того, процедуры, обладающие свойством повторности, могут «противоречить» сами себе, когда устранение одной неоптимальной конструкции порождает другую неоптимальную конструкцию. В качестве примера можно привести процедуру преобразования рекурсивно-определяемых переменных [9], повторную по отношению к процедуре уменьшения числа обращений к индексируемым переменным [7].

Так как в настоящей книге рассматриваются методы оптимизации применительно к однопроцессорным ЭВМ, то и все описываемые процедуры ориентированы на применение к программам, выполняемым на таких ЭВМ. Однако многие из описываемых процедур могут быть применены к программам, выполняемым на многопроцессорных ЭВМ. Выделение таких процедур требует специального анализа, который здесь не проводится, так как это выходит за рамки вопросов, излагаемых в книге.

В качестве иллюстрационного языка используется в основном ФОРТРАН, а в некоторых случаях для простоты изложения материала ПЛ/1, КОБОЛ, MINI-BASIC [72] и достаточно простой, легко понимаемый обобщенный язык ассемблера (в соответствующих местах указывается применяемый язык).

Многие неоптимальные конструкции, встречающиеся в программах, записанных на языках различных уровней, несмотря на их различную запись являются семантически эквивалентными. Поэтому значительная часть описываемых оптимизационных процедур применима (быть может с небольшими изменениями) не только к программам, записанным на иллюстрационном языке, но и к программам, записанным на других языках, в частности языке ассемблера. Если же эффект от выполнения процедуры основывается на особенностях одного конкретного языка, то на это указывается при описании такой процедуры.

Все оптимизационные процедуры разделены условно на группы, для облегчения изложения материала.

2. ОПТИМИЗАЦИЯ ЦИКЛОВ

Время выполнения программы в значительной мере зависит от времени выполнения многократно повторяемых участков, составляющих зоны или их наиболее употребимый частный случай — циклы. Незначительные потери времени, вызванные неэффективным построением зоны, могут значительно ухудшить качество всей программы. Поэтому в системах оптимизации программ всегда присутствуют процедуры, оптимизирующие зоны. (Так как в настоящее время наиболее полно исследованы процедуры, преобразующие циклы, то все описанные ниже оптимизационные процедуры ориентированы на циклы, хотя их можно легко распространить и на зоны).

Чистка циклов. Процедура чистки циклов уменьшает время выполнения цикла путем удаления из его тела арифметических выражений или их частей, не зависящих от управляющей переменной цикла. Зависимость от управляющей переменной может быть как явной (т. е. выражение содержит идентификатор управляющей переменной), так и неявной (т. е. выражение содержит переменные, в свою очередь, зависящие явно или неявно от управляющей переменной). Кроме того, в теле цикла могут присутствовать рекурсивно-определяемые переменные, которые изменяются при каждом выполнении цикла, т. е. также неявно зависящие от управляющей переменной.

Выполнение процедуры чистки циклов состоит из следующих четырех операций *: префиксации операторов присваивания;

* Под префиксацией структуры A относительно структуры B понимается размещение в программе структуры A до структуры B, т. е. при выполнении

префиксации арифметических подвыражений; постфриксации операторов присваивания; замены операций деления.

Рассмотрим, например, циклы

```
а) DO 1 K = 2, 50, 2
   B = A + C 1
   C = C + B
   E (K) = SIN (B) + A ** 2 + K/D
   N = K * K
   N 1 = N * 0. 2 + A
1  CONTINUE
```

```
б) B = A + C 1
   Y 1 = SIN (B) + A ** 2
   Y 2 = 1./D
   DO 1 K = 2, 50, 2
   C = C + B
   E (K) = Y 1 + K * Y 2
1  CONTINUE
   N 1 = 500. + A
```

причем цикл участка (б) является результатом выполнения процедуры чистки циклов над циклом участка (а). Здесь предполагается, что для переменной *N* выполняются все условия, позволяющие применить процедуру подстановки [12]. Время счета программы уменьшается вследствие того, что вынесенные операторы выполняются один раз (если отвлечься от случая вложенных циклов), а не *M* раз, где *M* — число повторений цикла, что при больших *M* весьма существенно. В приведенном примере *M* равно 25.

Если имеется несколько вложенных циклов, то префиксация и постфиксация арифметических подвыражений и операторов осуществляются как относительно внутреннего цикла, так и относительно циклов, его объемлющих. Для вложенных циклов рассматриваемая процедура выполняется последовательно: первоначально префиксация и постфиксация арифметических подвыражений и операторов осуществляются относительно самого внутреннего цикла, затем относительно цикла, его объемлющего, и т. д. Участок (г) является результатом выполнения процедуры чистки циклов над участком (в):

```
в) DO 1 I = 1, 20
   DO 1 J = 1, 20
   B (J) = B (J) + A (I)/D ** 3
1  CONTINUE
```

```
г) Y 1 = 1./D ** 3
   DO 1 I = 1, 20
   Y 2 = A (I)*Y1
   DO 1 J = 1, 20
   B (J) = B (J) + Y 2
1  CONTINUE
```

Здесь элемент массива A (I) рассматривается как арифметическое подвыражение, использующее управляющую переменную I внешнего цикла. Процедура чистки циклов для участка (в) выполняется в два этапа. На первом этапе выполняется префик-

---

программы сначала выполняется структура *A*, а затем структура *B*. Под постфиксацией структуры *A* относительно структуры *B* понимается размещение в программе структуры *A* за структурой *B*, т. е. при выполнении программы сначала выполняется структура *B*, а затем структура *A*. В соответствующем контексте, однозначно определяющем сказанное, употребляются выражения «префиксация структуры» и «постфиксация структуры». Структурами могут быть любые участки программы или отдельные операторы.

сация подвыражения A(I)/D* *3 относительно внутреннего цикла, а на втором этапе выполняется префиксация подвыражения D* * 3 относительно внешнего цикла. Участок (г) является результатом выполнения этих двух этапов над участком (в).

Следует заметить, что префиксация и постфиксация операторов и/или арифметических подвыражений относительно цикла не всегда приводят к сокращению числа их выполнения, т. е. к сокращению времени выполнения цикла, а, наоборот, могут увеличить время его выполнения. Такая ситуация возникает при некоторых, как правило, граничных значениях параметров, определяющих условия выполнения цикла. Рассмотрим, например, участок (д) (язык ПЛ/1), применение к которому процедуры чистки циклов может привести к увеличению времени его выполнения. Это увеличение возникает вследствие того, что цикл при конкретных параметрах, например при X (J) $\neq$ 0 или J $>$ K, определяющих условия его выполнения, не выполняется:

```
д) DO I = J TO K WHILE (X (I) = 0);
   X (I) = Y (I) * SQRT (N);
   END;
```

В современных системах оптимизации программ обнаружение таких ситуаций приводит к автоматическому блокированию выполнения процедуры чистки циклов.

Объединение циклов. При трансляции любого цикла с языка высокого уровня в машинные коды в объектной программе появляются операции, реализующие заголовок цикла, т. е. увеличение управляющей переменной цикла на значение его шага; сравнение управляющей переменной с граничным значением; передача управления. Процедура объединения циклов производит соединение нескольких циклов с одинаковыми заголовками в один. За счет этого уменьшается в несколько раз (в зависимости от числа объединяемых циклов) число выполнений команд, реализующих заголовок цикла, что сокращает время выполнения всей программы. Рассматриваемая процедура выполняется в том случае, когда перемещение одного из циклов в программе не меняет ее смысла. Например, участок (б) является результатом выполнения рассматриваемой процедуры над участком (а):

```
а) DO 1 K = 1, 100          б) DO 1 K = 1, 100
 1 A (K) = B (K) + K           A (K) = B (K) + K
   DO 2 L = 1, 100           1 E (K) = K * K
 2 E (L) = L* L
```

Основным параметром, который определяет возможность объединения, является шаг цикла. Процедура выполнима при условии его совпадения у рассматриваемых циклов. Если граничные значения управляющих переменных циклов не равны, то объединение осуществляется по совпадающим значениям управляющей

переменной цикла. Например,

в)
```
DO 2 K = 2, 100, 2
2 Y (K) = B (K) * K + S (K)/K
  ⋮

DO 3 L = 2, 150, 2
3 E (L) = L * * 2 * R (L)
```

г)
```
DO 2 K = 2, 100, 2
  Y (K) = B (K) * K + S (K)/K
2 E (K) = K * * 2 * R (K)
  ⋮
DO 3 L = 102, 150, 2
3 E (L) = L * * 2 * R (L)
```

Здесь (в) — первоначальный вид участка программы, (г) — результат выполнения процедуры. Получаемый выигрыш времени в этом случае зависит от числа совпадающих значений управляющих переменных циклов.

Помимо прямого оптимизационного эффекта выполнение рассматриваемой процедуры приводит также к косвенному оптимизационному эффекту, вызванному тем, что в ряде случаев объединение циклов создает возможность для выполнения процедуры экономии памяти при обработке массивов (см. гл. 2.9), которая осуществляет замену некоторых массивов одним скаляром.

Псевдообъединение циклов. Суть этой процедуры основана на некоторых особенностях процесса трансляции, учет которых позволяет сократить число выполнений операций, реализующих заголовок цикла. Действия, выполняемые процедурой псевдообъединения циклов, аналогичны в некотором смысле действиям, выполняемым процедурой объединения циклов (что и объясняет название процедуры), и могут быть интерпретированы как соединение нескольких циклов в один. Например, участок (б) является результатом выполнения процедуры псевдообъединения циклов над участком (а):

а)
```
DIMENSION A (30), B (50)
  ⋮
DO 1 I = 1, 30

1 A (I) = 3. 7
  ⋮
DO 2 J = 1, 50
2 B (J) = 3. 7
```

б)
```
DIMENSION A (30), B (50)

COMMON A, B
  ⋮
DO 1 I = 1, 80

1 A (I) = 3. 7
```

При выполнении процедуры псевдообъединения циклов учитывается тот факт, что после трансляции программы и загрузки ее в память два массива оказываются расположенными один за другим. Следует учитывать, что преобразования, выполняемые рассматриваемой процедурой, являются согласно некоторым стандартам нарушением правил языка ФОРТРАН.

Разделение циклов. В некоторых ЭВМ циклы реализуются в памяти специального вида (например, в стеках или быстрых регистрах), обеспечивающей их быстрое выполнение. Объем та-

кой памяти, как правило, невелик, что, в свою очередь, ограничивает объем эффективно выполняемых циклов. В этих случаях более эффективно заменить цикл, объем которого превосходит объем специальной памяти, двумя или более небольшими циклами, объем каждого из которых не превосходит объема специальной памяти.

Процедура разделения циклов, по своим действиям противоположная процедуре объединения циклов, состоит в замене цикла, для размещения которого требуется объем памяти, превосходящий объем памяти специального вида, эквивалентной ему последовательностью циклов, умещающихся в памяти специального вида. Например, участок (б) является результатом выполнения рассматриваемой процедуры над участком (а):

```
а) DO 1 I = 1, 100
   A (I) = I * 3
   C (I) = I * * 2

   S = S + I
   D (I) = A (I)/S + C (I) * S
1  CONTINUE
```

```
б) DO 1 I = 1, 100
   A (I) = I * 3
   C (I) = I * * 2

1  CONTINUE
   DO 2 I = 1, 100
   S = S + I
   D (I) = A (I)/S + C (I) * S
2  CONTINUE
```

При выполнении рассматриваемой процедуры выигрыш, получаемый за счет применения специальной памяти, превосходит проигрыш, возникающий вследствие увеличения числа выполнений команд, реализующих заголовок цикла.

Выполнение процедуры разделения циклов имеет также смысл при страничной организации памяти. Циклы большого объема могут располагаться на двух или нескольких страницах памяти, что увеличивает время выполнения цикла за счет организации переходов с одной страницы на другую. Разделение одного большого цикла на несколько небольших позволяет разместить каждый из этих циклов на одной странице памяти. Это приводит к тому, что суммарное время выполнения небольших циклов меньше времени выполнения эквивалентного им большого цикла.

Вынос ветвлений из цикла. Действия, выполняемые рассматриваемой процедурой, также в некотором смысле противоположны действиям, выполняемым процедурой объединения циклов.

Пусть исходный цикл состоит из двух частей, и в зависимости от выполнения определенного условия выполняется одна из этих частей. Тогда исходный цикл можно заменить двумя циклами, соответствующими двум частям исходного цикла. Например, участок (б) (язык ПЛ/1) является результатом выполнения процедуры выноса ветвлений из цикла над участком (а):

```
а) DO I = 1 TO 20;
   IFK = 2 THEN A (I) = B (I) * C (I);
   ELSE A (I) = B (I) + 2.;
   END;
```

```
б) IFK = 2 THEN
   DO I = 1 TO 20;
   A (I) = B (I) * C (I);
   END;
   ELSE
   DO I = 1 TO 20;
   A (I) = B (I) + 2.;
   END;
```

В результате выполнения процедуры выноса ветвлений из цикла уменьшается время счета программы вследствие сокращения числа проверок выполнения условий.

Развертка циклов. Цикл является одной из конструкций, использование которой сокращает объем памяти, занимаемый программой, но одновременно увеличивает время ее счета, так как при трансляции циклов в объектные коды появляются такие дополнительные операции, как изменение значения управляющей переменной, сравнение ее значения с заданным числом и передача управления. Если критерием оптимизации является быстродействие, то возможно преобразование программы, при котором из объектной программы, полученной в результате трансляции, устраняются команды, выполняющие упомянутые выше операции. Процедура, осуществляющая это преобразование, называется разверткой циклов. Суть ее заключается в замене управляющей переменной ее значениями. Замена может быть частичной или полной, по одной или нескольким управляющим переменным. В зависимости от характера использования управляющей переменной в теле цикла развертка цикла может осуществляться двумя способами, показанными участками (б) и (в), полученными в результате выполнения процедуры развертки циклов над участком (а):

```
а) DO 2 I = 1, 100
2  K (I) = I
```

```
б) DO 2 I = 1, 99, 2
     K (I) = I
   2 K (I + 1) = I + 1
```

```
в) DO 2 I = 1, 99, 2
     K (I) = I
     J = I + 1
   2 K (J) = J
```

Назовем степенью развертки количество копий цикла, появляющихся в программе в результате выполнения процедуры. Например, участок (а) — исходный цикл, участки (б) и (в) — результат развертки цикла со степенью развертки, равной двум. Полная развертка — это замена цикла последовательностью копий его тела, в которых идентификатор управляющей переменной заменен ее значением.

Если имеется несколько вложенных циклов, то аналогично

осуществляется развертка по каждой из управляющих переменных или по нескольким одновременно. Чем медленней меняются значения управляющей переменной в цикле, тем значительней эффект, получаемый в результате развертки цикла по этой переменной. Во избежание появления отрицательного оптимизационного эффекта и изменения смысла преобразованной программы процедуру развертки циклов следует применять только к циклам, у которых известно число их повторений. Процедура развертки циклов принадлежит к тому типу процедур, у которых четко просматривается «противоречие» между оптимизацией по времени выполнения программы и оптимизацией по занимаемой ею объему памяти.

Удаление пустых циклов. Суть этой процедуры состоит в удалении из программы пустых циклов, т. е. циклов, имеющих вид

```
DO 1  I = 1, 100
1 CONTINUE
```

Такие циклы могут появиться в программе в результате небрежности программиста, а чаще всего в результате выполнения оптимизационных и других преобразований. Следует, однако, отметить, что в некоторых случаях пустые циклы используются как специальный прием программирования. Например, на приведенном участке (язык ПЛ/1)

```
DO I = 1 TO K WHILE(X ¬= Y(I));
END;
IF I < K THEN
CALL CHECK 1;
ELSE
CALL CHECK 2;
```

Цикл используется для поиска элемента X в массиве Y (I) и определения возможности выполнения функции CHECK 1. Хотя это не самый лучший способ программирования подобных задач, такие участки встречаются в реальных программах. Заметим, что подобные приемы программирования могут встречаться при использовании языков, в которых значение управляющей переменной цикла сохраняется после его выполнения. Помимо использования в некоторых приемах программирования пустые циклы могут применяться в специальных программах, например, в программах реального времени для создания задержек. В общем случае выполнение процедуры устранения пустых циклов экономит время выполнения программы и объем занимаемой ею памяти.

Устранение циклов. Эта процедура применяется к вложенным циклам, таким, что во внутреннем цикле имеется некоторое условие, определяющее выполнение тела внутреннего цикла только в случае совпадения значений управляющих переменных вну-

треннего и внешнего циклов. Например, на участке (а) показан такой цикл (язык ПЛ/1), А и В — некоторые блоки:

```
а) DO I = K 1 TO K 2;
   A;
   DO J = N 1 TO N 2;
   IF J = I THEN
   B;
   END;
   END;
```

Действия, выполняемые оператором условного перехода IF, находящимся в теле внутреннего цикла, можно сформулировать следующим образом. Если области изменения управляющих переменных I и J пересекаются, и текущее значение переменной I принадлежит этой области, то выполняется блок В. Для вложенных циклов область пересечения управляющих переменных циклов определяется из соотношений

$$K1 \leqslant I \leqslant K2;$$
$$N1 \leqslant J \leqslant N2.$$

Объединив эти соотношения, получим следующую формулировку действий, выполняемых оператором условного перехода. Если выполнятся соотношения

$$K1 \leqslant N2;$$
$$N1 \leqslant K2;$$
$$MAX(K1,\ N1) \leqslant I \leqslant MIN(K2,\ N2),$$

то выполнится блок В. Как видно из этой формулировки, необходимость в организации внутреннего цикла отпадает. На этом свойстве основана процедура устранения циклов, результат выполнения которой над участком (а) показан на участке (б):

```
б) DO I = K 1 TO K 2;
   A;
   IF K 1 < = N 2 & N 1 < = K 2 & MAX (K 1, N 1) < = I & I <
   < = MIN (K 2, N 2)
   THEN DO;
   J = I;
   B;
   END;
   END;
```

В реальных программах параметры K1, K2, N1, N2 часто являются целыми константами. В этом случае в результате последовательного применения рассматриваемой процедуры и процедуры выполнения константных действий [8] возможны следующи два случая.

1. Области изменения индексов I и J пересекаются. Тогда оператор

```
IF K 1 < = N 2 & N 1 < = K 2 & MAX (K 1, N 1) < = I & I <
    < = MIN (K2, N2)
THEN DO;
```

преобразуется к виду

```
IF L 1 < = I & I < = L 2 THEN DO,
```

где L1 = MAX (K1, N1), L2 = MIN (K2, N2), и участок (б) принимает следующий вид:

```
в) DO I = K 1 TO K 2;
    A;
    IF L 1 < = I & I < = L 2 THEN DO;
    J = I;
    B;
    END;
    END;
```

2. Области изменения индексов I и J не пересекаются. Тогда условие

```
K 1 ≤ N 2 & N 1 ≤ K 2 & MAX (K 1, N 1) ≤ I & I ≤ MIN (K 2, N 2)
```

для любых значений индексов I и J всегда является ложным и оператор

```
IF K 1 < = N 2 & N 1 < = K 2 & MAX (K 1, N 1) = I & I <
    < = MIN (K 2, N 2)
THEN DO
```

преобразуется в оператор безусловного перехода, что эквивалентно невыполнению внутреннего цикла, т. е. участок (а) принимает следующий вид:

```
г) DO I = K 1 TO K 2;
    A;
    END;
```

Как видно из приведенных примеров, последовательное применение процедур устранения циклов и выполнения константных действий в случае, когда параметры K1, K2, N1, N2 являются целыми константами, позволяет получить дополнительный оптимизационный эффект, определяемый значениями параметров K1, K2, N1, N2. Кроме того, если управляющая переменная внутреннего цикла [для участков (а), (б), (в) — переменная J] употребляется в теле внутреннего цикла [для участков (а), (б), (в) — в блоке B] только локальным образом, то над телом внутреннего цикла можно выполнить оптимизационную процедуру отождествления переменных [8] и удалить из программы дополнительно оператор присваивания (для участков (б), (в) — оператор J = I). В результате выполнения процедуры отождествления переменных участок (в) преобразуется в участок (д):

```
д) DO I = K1 TO K2;
   A;
   IF L1 <= I & I <= L2 THEN DO;
   B;
   END;
   END;
```

Оптимизационный эффект рассматриваемой процедуры обусловлен удалением из программы операций изменения управляющей переменной цикла, сравнения ее значения с граничным и передачи управления.

Сдвиг циклов. Процедура сдвига циклов, используя возможности языков высокого уровня, выполняет сдвиг области изменения управляющей переменной цикла таким образом, чтобы верхняя граница изменения этой переменной равнялась нулю. Такие преобразования сокращают число команд, обеспечивающих изменение управляющей переменной цикла, так как ЭВМ, как правило, имеют команды, позволяющие осуществлять сравнение значения величины с нулем схемно или достаточно простым способом. Выполнение процедуры сдвига циклов приводит к изменению в теле цикла всех обращений к управляющей переменной цикла. Если значение управляющей переменной цикла используется после выхода из цикла, то в программе появляется дополнительный оператор присваивания, обеспечивающий совмещение преобразованного цикла с остальной программой. Например (язык ПЛ/1), участок (б) является результатом выполнения процедуры сдвига циклов над участком (а):

```
а) DO I = 1 BY 1 TO N;
   A(I) = B(I) + I * K;
   END;
```

```
б) DO I = 1 — N BY 1 TO 0;
   J = I + N;
   A(J) = B(J) + J * K;
   END;
   I = J;
```

В участке (б) появились два дополнительных оператора по сравнению с исходным участком (а), однако, если значение управляющей переменной цикла не используется вне цикла, второй из этих операторов можно удалить.

Существует другой вариант преобразований, выполняемых рассматриваемой процедурой. Например, участок (в) также является результатом выполнения процедуры сдвига циклов над участком (а):

```
в) DO I = 1 — N BY 1 TO 0;
   A(I + N) = B(I + N) + (I + N) * K;
   END;
   I = I + N;
```

Оптимизационный эффект процедуры сдвига циклов в рассматриваемом варианте и, следовательно, целесообразность применения одного из вариантов преобразований существенно зависят от структуры преобразуемого цикла, так как в результате выполнения этих преобразований появляются дополнительные вычисления, связанные с изменением значения управляющей переменной цикла. Заметим, что участок (в) можно получить путем выполнения процедуры усложнения индексных выражений [9] над участком (б).

Сжатие циклов. В ряде случаев в теле цикла встречаются условия, которые могут ограничивать область изменения управляющей переменной цикла. Процедура сжатия циклов осуществляет перенос таких ограничений в заголовок цикла. Например, участки (б) и (г) являются результатом выполнения рассматриваемой процедуры над участками (а) и (в) соответственно (здесь предполагается, что A < C < B (язык ПЛ/1)):

```
а) DO I = A TO B;
   IF I <= C THEN
   R(I) = P(I);
   END;

б) DO I = A TO C;
   R(I) = P(I);
   END;

в) DO I = A TO B;
   IF I => C THEN
   R(I) = P(I);
   END;

г) DO I = C TO B;
   R(I) = P(I);
   END;
```

В результате выполнения процедуры сжатия циклов из тела цикла удаляется логический оператор перехода и сокращается число выполнений тела цикла.

Преобразование рекурсивно-определяемых переменных. Эта процедура (называемая иначе уменьшением силы операций [81, 102]) в ряде случаев приводит к замене медленно выполняемых машинных команд более быстрыми. В зонах (циклах), содержащих арифметические выражения, использующие рекурсивно-определяемые переменные (рекурсия может быть задана явно или неявно), значение которых при каждом выполнении зоны (цикла) увеличиваются или уменьшаются на постоянную величину, существует возможность замены операции возведения в степень умножением и умножения — сложением. Например, участки (б) и (г) являются результатом выполнения рассматриваемой процедуры над участками (а) и (в) соответственно:

```
а) DO 1 K = 1, 100
   C(K) = A + B * K
   E(K) = T * D ** K
 1 CONTINUE

б) C(1) = A + B
   E(1) = T * D
   DO 1 K = 2, 100
   C(K) = C(K - 1) + B
 1 E(K) = E(K - 1) * D
```

```
в) DO 1 K = 1, 100
   R = R 1 * K + R 2
   F = P * G * * K
   CALL A 1 (R, F)
 1 CONTINUE
```

```
г) R = R 2
   F = P
   DO 1 K = 1, 100
   R = R + R 1
   F = F * G
   CALL A 1 (R, F)
 1 CONTINUE
```

Здесь операция умножения, которая повторяется в цикле, заменяется более простой операцией сложения, а операция возведения в степень — операцией умножения, что в сочетании с многократностью выполнения в цикле рассматриваемых операторов, в общем случае, дает выигрыш во времени. Необходимо отметить, что в участке (б) появились два лишних обращения к индексируемой переменной по сравнению с участком (а); это является примером такой ситуации, когда одна оптимизационная процедура противоречит другой (в данном случае процедуре уменьшения числа обращений к индексируемым переменным [9]). В подобных ситуациях необходим машинный эксперимент, на основе которого определяется целесообразность выполнения процедуры. Кроме того, при выполнении этой процедуры следует учитывать, что операция I * K точнее, чем операция сложения I + I + ...I.

Выполнение рассматриваемой процедуры особенно эффективно на уровне внутренних языков или автокодов при вычислении значений индексов, которые, как правило, являются рекурсивно-определяемыми переменными.

Замена проверок условия. Часто выход из зоны (и, как правило, всегда из цикла) осуществляется по достижению текущего значения некоторой переменной (для циклов управляющей переменной) граничного значения. В таких случаях один из операторов зоны, как правило, последний, сравнивает текущее значение такой переменной с ее граничным значением и определяет возможность повторного выполнения зоны. Пусть в зоне имеются рекурсивно-определяемые переменные, значения которых увеличиваются на постоянную величину при каждом выполнении зоны. Суть процедуры замены проверок состоит в устранениии из зоны операций изменения текущего значения переменной, определяющей число выполнений зоны, и в замене операции сравнения этого значения с граничным значением на операцию сравнения текущего значения рекурсивно-определяемой переменной с ее значением при последнем выполнении зоны. Например, участок (б) является результатом выполнения процедуры над участком (а):

```
а) M = 0
   I = 0
 1 I = I + 1
   CALL A (M, X 2)
```

```
б) M = 0
   M 2 = N 1 * 10
 1 CALL A (M, X 2)
   M = M + N 1
```

```
M = M + N1                    IF (M. LT. M2) GOTO 1
IF (I. LT. 10) GOTO 1
```

Уменьшение времени выполнения программы и объема занимаемой ею памяти обусловлено удалением оператора I = I + 1 из зоны.

Рассматриваемая процедура выполняется аналогичным образом, если в зоне имеются рекурсивно-определяемые переменные, значения которых увеличиваются в одинаковое число раз при каждом выполнении зоны. Например, участок (г) является результатом выполнения процедуры замены проверок условия над участком (в):

```
в)  I = 0                     г)  F = P
    F = P                         F1 = P * G ** 15
  1 I = I + 1                   1 F = F * G
    F = F * G                     CALL B (F)
    CALL B (F)                    IF (F. LT. F1) GOTO 1
    IF (I. LT. 15) GOTO 1
```

Наиболее часто возможность выполнения рассматриваемой процедуры возникает после выполнения процедуры преобразования рекурсивно-определяемых переменных. В частности, участки (а) и (в) получены в результате выполнения процедуры преобразования рекурсивно-определяемых переменных над участками (д) и (е) соответственно:

```
д)  I = 0                     е)  I = 0
  1 I = I + 1                   1 I = I + 1
    CALL A (M, X2)                F = P * G ** I
    M = N1 * I                    CALL B (F)
    IF (I. LT. 10) GOTO 1         IF (I. LT. 15) GOTO 1
```

3. ОПТИМИЗАЦИЯ ИНДЕКСНЫХ ВЫРАЖЕНИЙ

Поиск элементов массива по значениям его индексов, как правило, требует значительных затрат времени. Это связано с тем, что внутреннее расположение элементов массива отличается от внешнего, многомерные массивы преобразуются в одномерные. При обращении к элементу массива выполняются две основные операции: преобразование списка индексов к каноническому виду; поиск элемента массива по каноническому виду списка индексов. При правильном написании индексных выражений время доступа к элементу массива можно свести к минимуму.

Так как в литературе использованы для обозначения различных понятий одинаковые термины, то примем следующее: элемент массива будем называть индексируемой переменной; арифметическое выражение (или выражения в многомерных массивах), определяющее номер элемента массива,— индексным выражением; переменные, являющиеся операндами индексных выраже-

ний,— индексными переменными. Если индексное выражение является тривиальным, т. е. является переменной, то эту переменную будем называть индексом. Например, в выражении A (I + 3, J, K) A — индексируемая переменная; I + 3 — индексное выражение; I — индексная переменная; J, K — индексы.

Перестановка индексов. В языках высокого уровня (в том числе в ФОРТРАНе) предполагаются вполне определенное расположение (упорядочивание) элементов многомерных массивов и связанный с этим порядок изменения индексов при построении циклов. Так, в ФОРТРАНе двумерные массивы (матрицы) записываются в память «развернутыми» по столбцам, т. е. первый индекс является наиболее быстро изменяющимся.

При построении цикла следует учитывать это и записывать элементы массива таким образом, чтобы порядок индексов и индексируемой переменной, пробегающей по элементам многомерного массива, соответствовал порядку этих же индексов, употребляемых в качестве управляющих переменных вложенных циклов. Наиболее быстро изменяющемуся индексу должен соответствовать внутренний цикл. Процедура перестановки индексов производит указанное упорядочение и может существенно уменьшить время выполнения программы. Например, участок (б) является результатом выполнения процедуры перестановки индексов над участком (а):

```
а) DO 2 K = 1, 100        б) DO 2 K = 1, 100
   DO 2 I = 1, 100           DO 2 I = 1, 100
2 N (K, I) = I * K        2 N (I, K) = I * K
```

Здесь участок (б) выполняется быстрее, чем участок (а).

В случае, если в ЭВМ принята страничная организация памяти, выполнение процедуры перестановки индексов приводит к дополнительному оптимизационному эффекту. Это связано с тем, что большие массивы могут располагаться на разных листах памяти, и при неправильном обращении к элементам таких массивов увеличивается число переходов с одного листа памяти на другой. Для ФОРТРАНа «развернутая» запись массивов приводит к тому, что элементы массивов, расположенные в одной строке, но в разных столбцах, после «развертывания» могут быть расположены в разных концах такой записи. Это при обращении к элементам массивов по строкам приводит к значительному увеличению числа переходов с одной страницы на другую. Выполнение процедуры перестановки индексов сводит число таких переходов к минимуму, что приводит к дополнительному сокращению суммарного времени обращения к элементам массивов.

Приведение индексных выражений к канонической форме. Особенности «индексной» арифметики в большинстве ЭВМ таковы, что для сокращения времени выполнения индексного выраже-

ния целесообразно во входном языке записывать его в одной из следующих (канонических) форм: V; C; V + C; C * V; C1 * V ± ± C2, где C, C1 C2 — положительные целые константы; V — целочисленная переменная. Вместе с тем следует подчеркнуть, что трансляторы строятся так, что фрагмент будет транслироваться и выполняться правильно при алгебраически эквивалентной записи индексных выражений, например C ± V; V * C; C2 ± V * C1 и др. Рассматриваемая процедура осуществляет преобразование индексных выражений к одной из указанных канонических форм. Учитывая, что вычисление индексных выражений, как правило, производится внутри циклов, выполнение процедуры приведения индексных выражений к канонической форме может дать значительный эффект.

Уменьшение числа обращений к индексируемым переменным. Каждому обращению к индексируемой переменной в программе, записанной на входном языке, соответствует группа машинных команд (в частности, команд «индексной» арифметики), вычисляющая значение индексного выражения, а затем адрес искомого элемента массива. Таким образом, крайне нежелательно любое неоправданное обращение к индексируемой переменной, тем более содержащейся в теле цикла. Процедура уменьшения числа обращений к индексируемым переменным предотвращает лишние обращения к одной и той же индексируемой переменной. Это осуществляется следующим образом. Значение индексируемой переменной, к которой многократно обращаются, присваивается специально вводимой в программу переменной. Затем в тексте программы рассматриваемая индексируемая переменная заменяется введенной переменной. В результате во время выполнения программы происходит только одно обращение к рассматриваемой индексируемой переменной:

```
а) M = A (10 * I — 25) + 34
   N = M + 2 * A (10 * I — 25) + A (10 * I — 25) * * 2
б) Y 1 = A (10 * I — 25)
   M = Y 1 + 34
   N = M + 2 * Y 1 + Y 1 * * 2
```

Здесь участок (б) является результатом преобразования участка (а) процедурой уменьшения числа обращений к индексируемым переменным.

Многократное обращение к индексируемой переменной возможно также вследствие того, что она находится в цикле. В этом случае при выполнении некоторых условий возможен вынос этой переменной за цикл. Например, на участке (в) индексируемая переменная D (I, J) находится в цикле и вследствие этого к ней происходит многократное обращение. На участке (г), являющемся результатом выполнения процедуры уменьшения числа обращений к индексируемым переменным над участком (в), перемен-

ная D (I, J) вынесена из цикла и к ней происходит обращение только один раз:

```
в) D (I, J) = 0
   DO 1 K = 1, 100
 1 D (I, J) = A (I, K) * B (K, J) + D (I, J)
г) DIJ = 0
   DO 1 K = 1, 100
 1 DIJ = A (I, K) * B (K, J) + DIJ
   D (I, J) = DIJ
```

Выполнение рассматриваемой процедуры уменьшает время выполнения программы и сокращает объем занимаемой ею памяти.

Усложнение индексных выражений. Вычисление индексных выражений часто производится с помощью специальной «индексной» арифметики в процессе трансляции, а не выполнения программы, и строится таким образом, чтобы максимально сократить его время. Поэтому необходимо рационально использовать аппаратные средства путем построения наиболее полных (для данного участка программы) конструкций выражений. Суть процедуры усложнения индексных выражений состоит в построении таких наиболее полных конструкций индексных выражений. Например, участки (б), (г), (е), при прочих равных условиях, выполняются быстрее участков (а), (в), (д) соответственно, причем наиболее ощутимая экономия времени получается в случае (е), так как здесь полностью используются возможности «индексной» арифметики:

```
а) M = I + 4                 в) M = 3 * I
   N = A (M) + K                N = A (M) + K
б) N = A (I + 4) + K         г) N = A (3 * I) + K
        д) M = 3 * I — 2
           N = A (M) + K
        е) N = A (3 * I — 2) + K
```

Построение наиболее полных конструкций индексных выражений увеличивает возможность применения других оптимизационных процедур, например приведения индексных выражений к канонической форме; уменьшения числа обращений к индексируемым переменным. Если обращение к индексируемой переменной происходит в цикле, то выполнение процедуры усложнения индексных выражений позволяет выразить наиболее явным образом зависимость индексного выражения от управляющей переменной цикла, что упрощает выполнение такой процедуры, как чистка циклов.

Сокращение числа вычислений индексных функций. При трансляции в машинные коды многомерные массивы, как правило, транслируются в одномерные, при этом индексируемые переменные $A\ (i_1,\ i_2,\ \ldots,\ i_k)\ k \geqslant 2$ преобразуются в элементы одномер-

ного массива $A\left(\prod_{j=1}^{k} S_j\right)$, где S_j — размерность массива по j-му измерению, т. е. происходит вычисление индексной функции, определяющей относительный адрес элемента массива. Под индексной функцией понимается эквивалент индексных выражений на уровне промежуточного языка. Индексные функции реализуются специальными подпрограммами и при записи программы на промежуточном языке или языке ассемблера явно присутствуют в ее тексте. Рассматриваемая процедура осуществляет сокращение числа вычислений индексных функций, т. е. сокращение числа вызовов соответствующих подпрограмм. Для сокращения числа вычислений индексных функций существуют две возможности:

1. Поиск в программе индексных функций, вычисляющих одинаковые относительные адреса элементов массивов, и вычисление таких адресов один раз. Например, в результате трансляции выражения A (J,I, K) = B (J, I, K) + C (J, I, K) в промежуточный язык появится три вычисления индексной функции, которые, если размерности массовов A, B, C одинаковы, можно заменить одним вычислением этой функции.

2. Представление индексных функций в виде линейной формы от ранее вычисленных индексных функций. Например, при трансляции оператора A (I, J) = A (I, J —1) ** 2 на промежуточном языке появляются два вычисления индексной функции, где второе из них, соответствующее элементу массива в правой части оператора, может быть представлено в виде линейной формы от индексной функции, соответствующей элементу массива в левой части оператора. Рассмотрим, например, преобразование процедурой сокращения числа вычислений индексных функций участка (б), являющегося внутренним представлением участка (а), где A, B, C, D — двумерные массивы размерности 10 × 10; AR, BR, CR, DR — внутренние представления этих массивов:

```
а) DO 1 I = 1, 9
   DO 1 J = 1, 9
   C (I, J) = A (I, J) * B (I, J)
   B (I, J) = D (I, J + 1) * A (J, I + 1)
 1 CONTINUE
б) DO 1 I = 1, 9
   DO 1 J = 1, 9
   CR [(I — 1) * 10 + J] = AR [(I — 1) * 10 + J)] * BR [(I — 1) *
                 * 10 + J]
   BR [(I — 1) * 10 + J] = DR [(I — 1) * 10 + J + 1] *
                 * AR [(J — 1) * 10 + I + 1]
 1 CONTINUE
```

В результате выделения одинаковых индексных функций, раскрытия скобок в индексных функциях и выноса не зависимых

от переменной J частей этих функций из внутреннего цикла участок (б) преобразуется в участок (в):

```
в) DO 1 I = 1, 9
     K4 = 10 * I — 10

   K3 = I — 9
   DO 1 J = 1, 9
   K1 = K4 + J
   CR[K1] = AR[K1] * BR[K1]
   BR[K1] = DR[K1 + 1] * AR[10 * J + K3]
 1 CONTINUE
```

Следует отметить, что хотя выполнение процедуры сокращения числа вычислений индексных функций можно свести к последовательному выполнению ряда других оптимизационных процедур (таких, как оптимизация выражений, чистка циклов и т. д.), здесь она выделена как самостоятельная процедура. Это обусловлено тем, что, как правило, процедура сокращения числа вычислений индексных функций реализуется в системах оптимизации программ отдельным блоком, а составляющие ее алгоритмы ориентированы на выполнение в контекстных условиях, связанных с индексными функциями. Более того, составляющие ее алгоритмы реализуются в таких случаях как одно целое и неотделимы друг от друга.

Использование различных индексных переменных. В ряде языков программирования, например в ФОРТРАНе, элементы многомерных массивов во внутреннем языке представляются в виде линейной последовательности. При обращении к элементам таких массивов происходит вычисление индексных функций, на основе которых определяется относительный адрес этих элементов в данном массиве. Принципы построения трансляторов таковы, что, как правило, вычисляются сразу все индексные функции, зависящие от данной индексной переменной. Для вычисления индексных функций используются быстрые регистры, число которых в ЭВМ ограничено, что может приводить к замедлению вычислений индексных функций. Суть рассматриваемой процедуры заключается в определении различным индексируемым переменным различных индексных переменных. В наиболее простом варианте при обработке массивов в цикле эта процедура сводится к определению циклам, обрабатывающим эти массивы, различных управляющих переменных. Например, участок (б) является результатом выполнения рассматриваемой процедуры над участком (а):

```
а) DO 1 J = 1, 100
 1 B(J + 5) = J * 5
   DO 2 J = 1, 50
 2 C(J * 3 + 2) = 0
```

```
б) DO 1 J = 1, 100
 1 B(J + 5) = J * 5
   DO 2 I = 1, 50
 2 C(I * 3 + 2) = 0
```

Процедура использования различных индексных переменных в некотором смысле противоречит различным процедурам экономии памяти, так как в программу вводятся дополнительные переменные, однако ее применение к участкам программы позволяет сократить время выполнения этих участков.

Уменьшение размерности обрабатываемых массивов. Вычисление индексных функций особенно неэффективно для многомерных массивов. Как правило, при составлении программ следует придерживаться принципа — никогда не использовать вектор, когда можно использовать скаляр, и никогда не использовать массив размера N, если есть возможность использовать массив размера $N - 1$. Наличие в ряде языков высокого уровня специальных средств, обеспечивающих совмещение величин в памяти ЭВМ, позволяет, следуя указанному принципу, заменять обработку многомерных массивов эквивалентной ей обработкой одномерных массивов. Суть процедуры уменьшения размерности обрабатываемых массивов состоит в замене элементов многомерных массивов, обрабатываемых в циклах, на совмещенные с ними по памяти элементы массивов меньших размеров. Например, участок (б) является результатом выполнения рассматриваемой процедуры над участком (а):

```
а) DIMENSION A(10, 10, 10)
   DO 1 I = 1, 10
   DO 1 J = 1, 10
   DO 1K = 1, 10
 1 A(K, J, I) = 0.
```

```
б) DIMENSION A(10, 10, 10), B(1000)
   EQUIVALENCE(A(1, 1, 1), B(1))
   DO 1 I = 1, 1000
 1 B(I) = 0.
```

Для ФОРТРАНа, используя его специфику, возможен другой вариант выполнения этой процедуры. В частности, участок (в) также является эквивалентным участку (а):

```
в) DIMENSION A(10, 10, 10)
   DO 1 I = 1, 1000
 1 A(I) = 0.
```

Аналогично, для языка MINI-BASIC участок (д) является результатом выполнения рассматриваемой процедуры над участком (г):

```
г) 10 FOR I = 1 TO 10
   11 FOR J = 1 TO 10
   12 LET A(I, J) = 0
   13 NEXT J
   14 NEXT I
```

```
д) 10 FOR I = 1 TO 100
   11 LET A(I) = 0
   12 NEXT I
```

На основании того свойства, что для ряда языков программирования массивы в ЭВМ располагаются по столбцам, рассматриваемая процедура выполняет замену обращения к отдельным столбцам многомерных массивов на обращение к одномерным массивам. Например, участок (з) является результатом выполнения процедуры уменьшения размерности обрабатываемых массивов над участком (е):

```
е) DIMENSION A (20, 20)
        :
        :
   DO 1 K = 1, 20
 1 A (4, K) = K * B
```

```
з) DIMENSION A (20, 20), R (20)
   EQUIVALENCE (R (1), A (61)
        :
        :
   DO 1 K = 1, 20
 1 R (K) = K * B
```

Оптимизационный эффект рассматриваемой процедуры возникает вследствие уменьшения числа вычислений индексных функций.

4. ОПТИМИЗАЦИЯ ЛОГИЧЕСКОЙ СТРУКТУРЫ ПРОГРАММЫ

Большинство оптимизационных процедур, связанных с изменением логической структуры программы, которые по характеру выполняемых преобразований наиболее близки к процедурам, осуществляющим преобразование алгоритмов и структурных схем программ, осуществляют глобальную оптимизацию, и для их выполнения необходим анализ управляющих и информационных связей программы.

Чистка гамаков. Процедура чистки гамаков состоит из двух преобразований — префиксации и постфиксации операторов относительно гамака. Преобразование префиксации операторов состоит в том, что если на любом пути в гамаке (т. е. на пути от начальной вершины гамака до его конечной вершины) содержится один из семантически эквивалентных операторов присваивания $a_1, a_2 \ldots, a_k$, $k \geqslant 2$ и соблюдается ряд условий, определяющих возможность выполнения рассматриваемого преобразования, то в начальную вершину гамака помещается оператор a_x, семантически эквивалентный операторам $a_1, a_2, \ldots, a_k$, а операторы $a_1, a_2, \ldots, a_k$ удаляются из программы. Основные условия, определяющие возможность этого преобразования, т. е. сохраняющие смысл программы, заключается в следующем:

1) на любом пути от начальной вершины гамака до операторов $a_1, a_2, \ldots, a_k$ не существует использований переменной, являющейся левой частью операторов $a_1, a_2, \ldots, a_k$;

2) переменные, используемые в правых частях операторов $a_1, a_2, \ldots, a_k$, не меняют своих значений на путях, начинающихся входной вершиной гамака и кончающихся одним из операторов $a_1, a_2, \ldots, a_k$.

Граф, показанный на рис. 10, *б*, является результатом выполнения преобразования префиксации операторов процедуры чистки гамаков, над графом, показанным на рис. 10, *а*. Незаштрихованные вершины этих графов образуют гамак, а операторы a_5, a_6, a_7, a_x являются семантически эквивалентными.

Алгоритм реализации преобразования префиксации операторов процедуры чистки гамаков имеет много общего с алгоритмом реализации процедуры оптимизации выражений. Поэтому целесообразно предварительное выполнение процедуры оптимизации выражений [11], а только затем — преобразования префиксации операторов рассматриваемой процедуры, так как это позволит упростить алгоритм реализации этого преобразования. Действительно, в этом случае будет выполняться префиксация самых простых операторов присваивания, т. е. таких, правая часть которых является переменной. Оптимизационный эффект преобразования префиксации операторов процедуры чистки гамаков определяется числом операторов присваивания, удаляемых из программы.

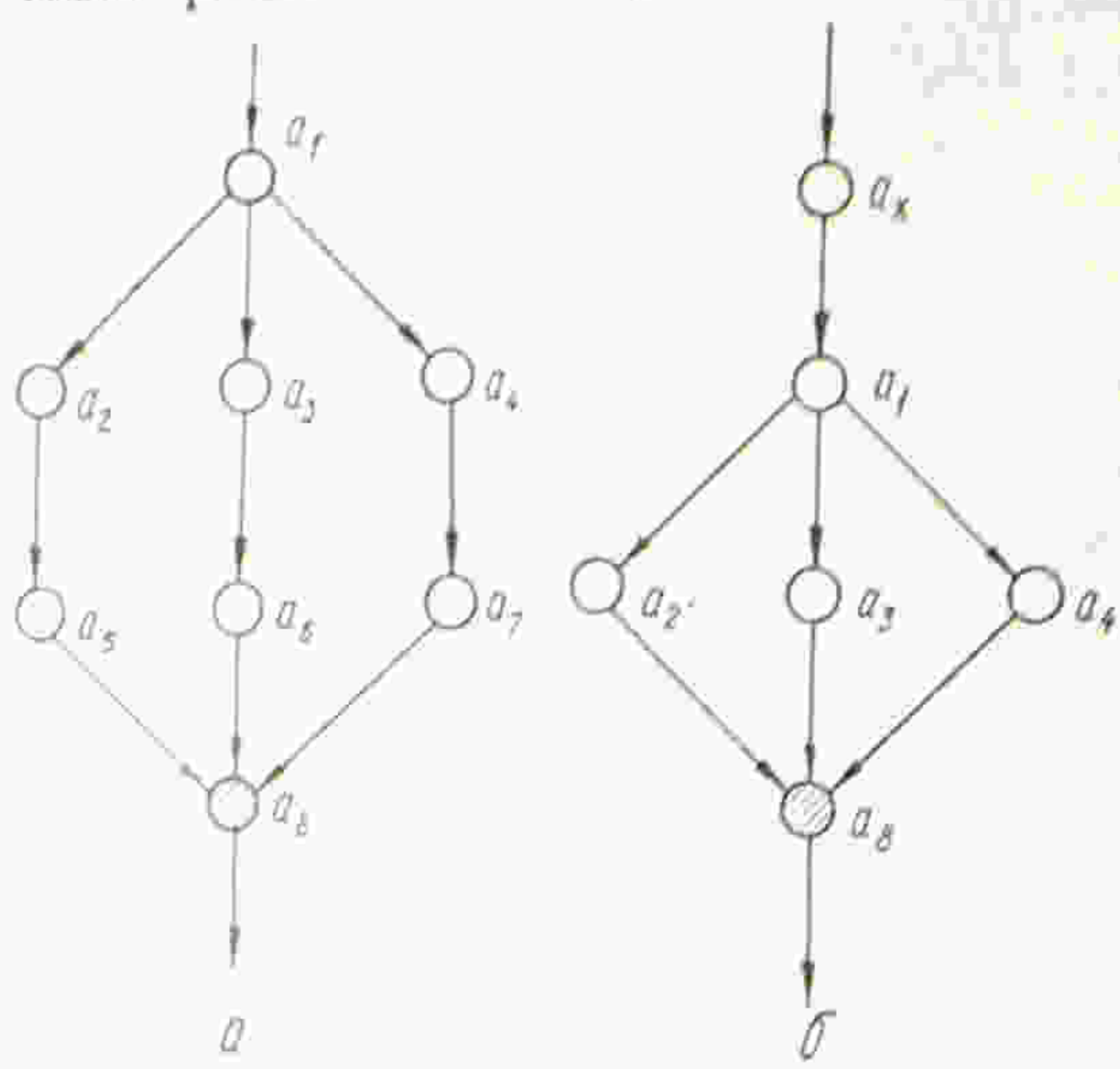


Рис. 10

Преобразование постфиксации операторов состоит в следующем. Пусть любой путь в гамаке содержит один из семантически эквивалентных операторов a_1, a_2, ..., a_k, $k \geqslant 2$. Если соблюдается ряд условий, определяющих возможность выполнения рассматриваемого преобразования, то операторы a_1, a_2, ..., a_k удаляются из программы, а перед конечной вершиной гамака помещается оператор a_x, семантически эквивалентный операторам a_1, a_2, ..., a_k. Основные условия, определяющие возможность этого преобразования, заключаются в следующем:

1) на любом пути от операторов a_1, a_2, ..., a_k до конечной вершины гамака не существует использований переменной, являющейся левой частью операторов a_1, a_2, ..., a_k;

2) переменные, используемые в правой части операторов a_1, a_2, ..., a_k, не меняют своего значения на путях, начинающихся одним из операторов a_1, a_2, ..., a_k и кончающихся конечной вершиной гамака.

Граф, показанный на рис. 11, *б*, является результатом выполнения преобразования постфиксации операторов процедуры чистки гамаков над графом, показанным на рис. 11, *а*. Незаштрихованные вершины этих графов образуют гамак, а операторы a_6, a_7, a_x являются семантически эквивалентными. Преобразо-

вание постфиксации операторов процедуры чистки гамаков позволяет также выносить в конечную вершину гамака части операторов. В таком варианте это преобразование в некоторой степени напоминает процедуру оптимизации выражений. На рис. 12, *б* показан граф, являющийся результатом выполнения преобразования постфиксации операторов процедуры чистки гама-

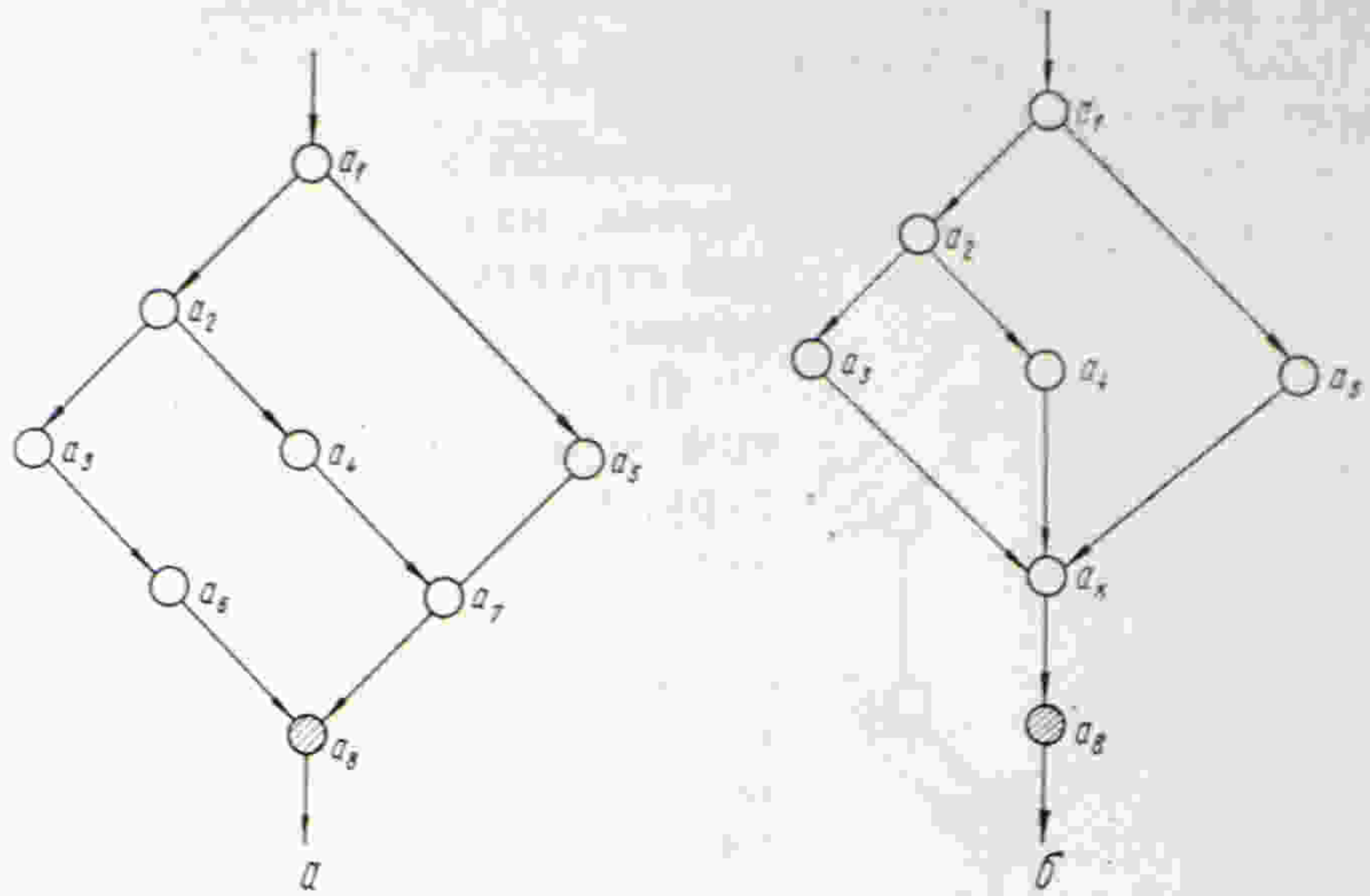


Рис. 11

ков над графом, показанным на рис. 12, *а*. Множество операторов a_2 и a_3 на рис. 12, *а*, имеющих вид

$$A\,1 = B * C ** 2 + E\,1$$
$$A\,1 = B * C ** 2 + E\,2,$$

соответственно преобразуется во множество операторов a_2, a_3 и a_x (рис. 12, *б*):

$$Y\,1 = E\,1$$
$$Y\,1 = E\,2$$
$$A\,1 = B * C ** 2 + Y\,1$$

соответственно, где Y1 — вводимая в программу дополнительная переменная. Операторы a_1, a_4, a_5 и a_6 имеют произвольный, но одинаковый на рис. 12, *а* и *б* вид. Незаштрихованные вершины этих рисунков составляют гамак.

Оптимизационный эффект преобразования постфиксации операторов процедуры чистки гамаков определяется числом операторов присваивания или арифметических подвыражений, удаляемых из программы.

Втягивание в гамак. Суть этой процедуры заключается в том, что некоторая структура программы с одним входом и одним выходом (такой структурой, в частности, может быть оператор, линейный участок или гамак) переносится назад по программе

в начальную вершину некоторого гамака, т. е. осуществляется префиксация этой структуры относительно гамака. Основные условия, определяющие возможность и целесообразность этого преобразования, следующие:

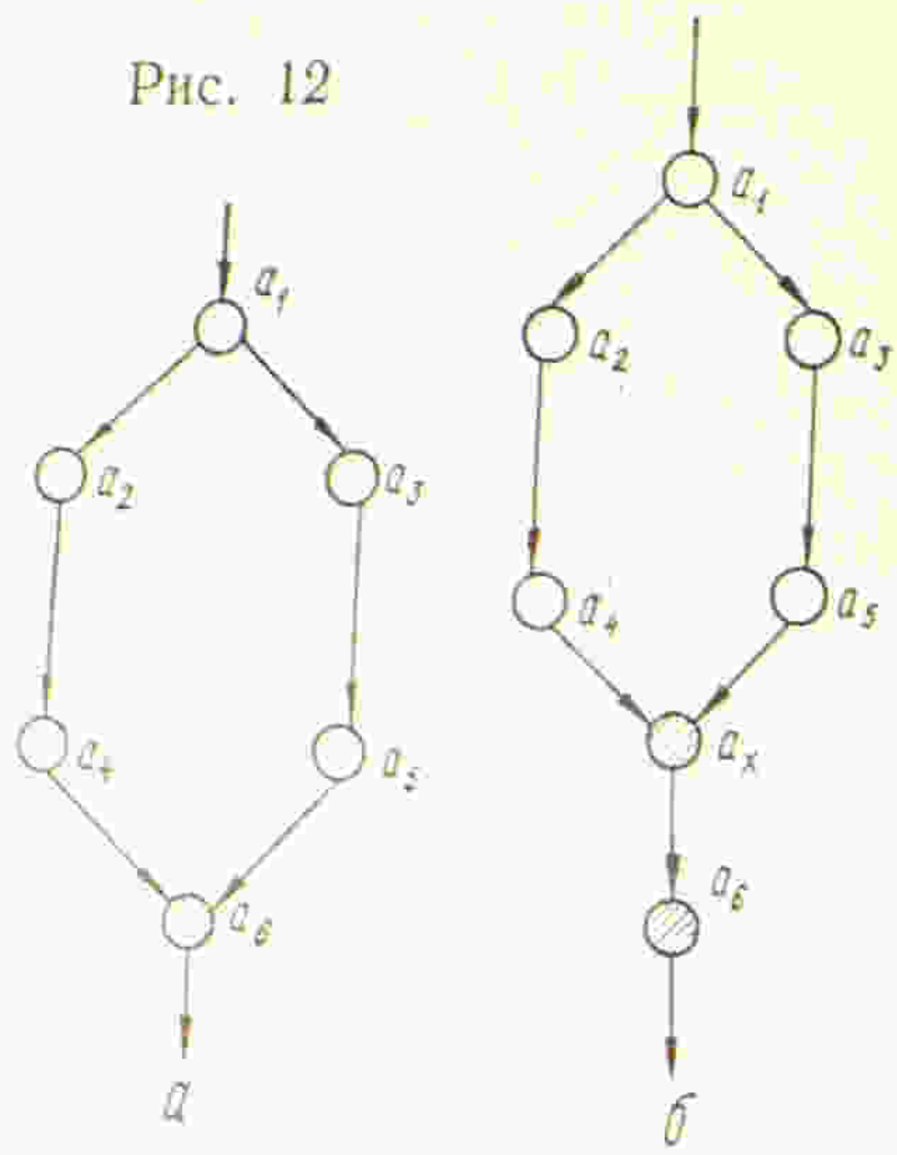
1) переменные, определяемые в перемещаемой структуре, используются только в гамаке и/или в операторах, лежащих на путях, выходящих из конечной вершины гамака;

2) любой путь, проходящий через гамак, обязательно проходит через перемещаемую структуру;

3) ни одна переменная, используемая в перемещаемой структуре, не переопределяется на путях, начинающихся перемещаемой структурой и кончающихся гамаком;

4) существуют пути, проходящие через рассматриваемую структуру, но не проходящие через гамак.

Рис. 12



Первые три условия определяют эквивалентность исходного и полученного в результате выполнения описанных преобразований участков программы, а четвертое условие определяет получение положительного оптимизационного эффекта в результате ее выполнения. Граф, показанный на рис. 13, *б*, является результатом выполнения процедуры втягивания в гамак над графом, показанным на рис. 13, *а*. Здесь в качестве структуры рассматривается оператор a_1, для которого предполагается выполнение перечисленных выше условий, а операторы a_4, a_5, a_6 составляют гамак.

Оптимизационный эффект, получаемый в результате выполнения рассматриваемой процедуры, обусловлен тем фактором, что любой путь, проходящий через гамак, проходит также через перемещаемую структуру, но существуют пути, проходящие через перемещаемую структуру и не проходящие через гамак. Поэтому в результате перемещения рассматриваемой структуры, для некоторых наборов данных, определяющих путь, проходящий через перемещаемую структуру и не проходящий через гамак, время выполнения программы сокращается на время выполнения рассматриваемой структуры.

Чистка фрагментов. Рассмотрим некоторый фрагмент, у которого операторы a_{n_j}, $j \in (\overline{1, J})$, $J \geqslant 1$ входные, и операторы a_{n_i}, $i \in (1, I)$, $I \geqslant 1$ выходные. Процедура чистки фрагментов состоит из двух преобразований — префиксации и постфиксации операторов относительно фрагмента. Так как фрагменты являются обобщением гамаков, то процедура чистки фрагментов в не-

котором смысле является обобщением процедуры чистки гамаков, а содержание преобразований — префиксации и постфиксации — операторов этих процедур практически одинаково. Однако наличие нескольких входов и выходов у фрагментов обусловливает существенные различия алгоритмов реализации этих процедур.

Граф, показанный на рис. 14, *б*, является результатом выполнения преобразования префиксации операторов процедуры «чистка фрагментов» над графом, показанным на рис. 14, *а*. Графы

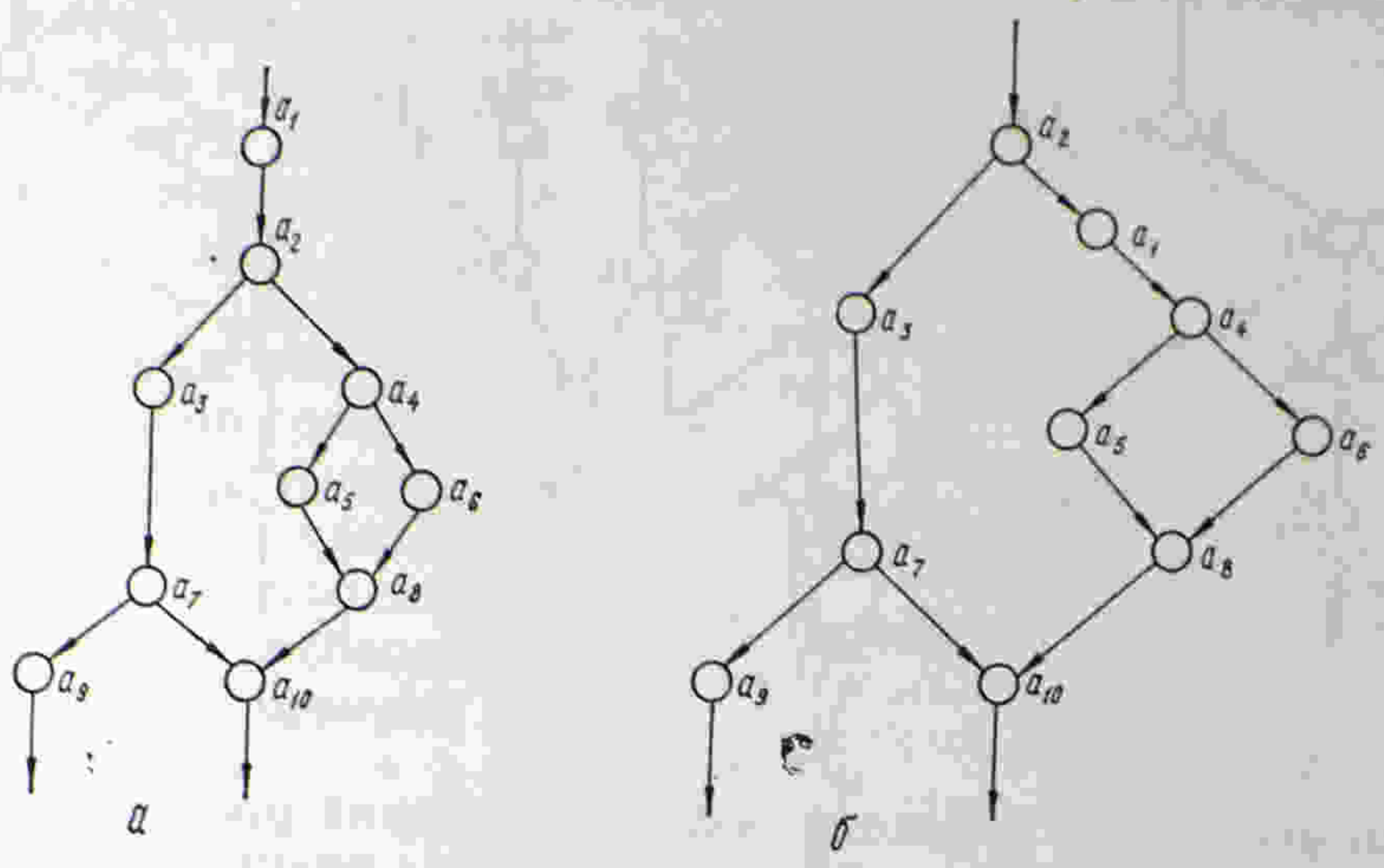


Рис. 13

на рис. 14, *а* и *б* являются фрагментами, а операторы a_4, a_5, a_6, a_{x_1}, a_{x_2} — семантически эквивалентны. Оптимизационный эффект преобразования префиксации операторов процедуры чистки фрагментов определяется разницей между числом операторов присваивания, удаляемых из программы, и числом операторов присваивания, вставляемых в программу перед входными операторами фрагмента.

Граф, показанный на рис. 15, *б*, является результатом выполнения преобразования постфиксации операторов процедуры чистки фрагментов над графом, показанным на рис. 15, *а*. Графы рис. 15, *а* и *б* являются фрагментами, а операторы a_4, a_5, a_6, a_{x_1}, a_{x_2} — семантически эквивалентны. Оптимизационный эффект преобразования постфиксации операторов процедуры чистки фрагментов определяется разницей между числом операторов присваивания, удаляемых из программы, и числом операторов присваивания, вставляемых в программу перед выходными операторами фрагмента.

Переупорядочение линейных участков. Использование операторов безусловного перехода позволяет из отдельных частей, в частности линейных участков, составлять программу, не заботясь о взаимном расположении этих частей в программе. Однако

использование операторов безусловного перехода приводит к дополнительным затратам ресурсов ЭВМ в процессе счета программы. Процедура переупорядочения линейных участков уменьшает число используемых операторов безусловного перехода за счет более эффективного расположения линейных участков в программе. Например, участок (*б*) является результатом выполнения процедуры переупорядочения линейных участков над участком (а)

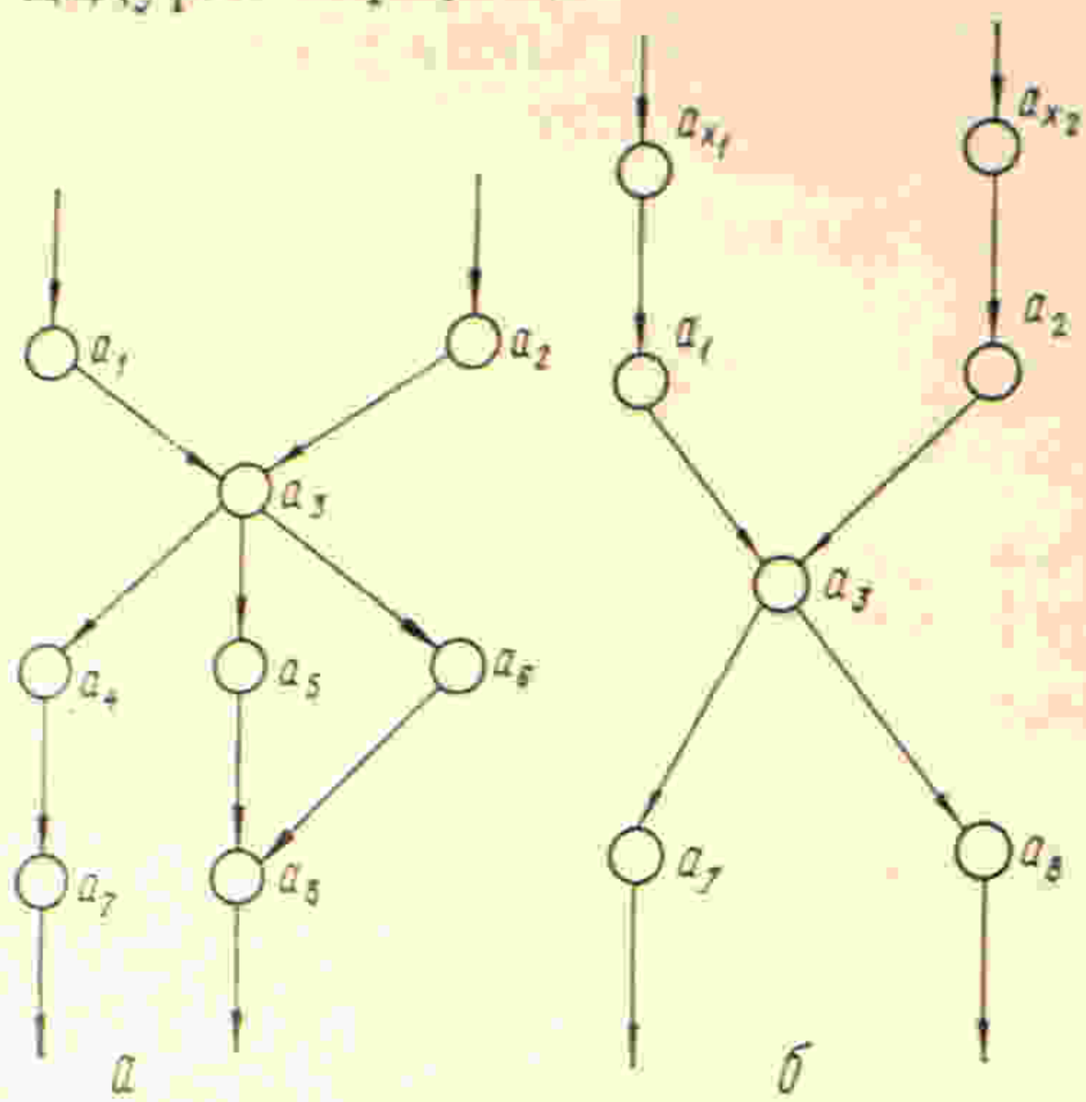


Рис. 14

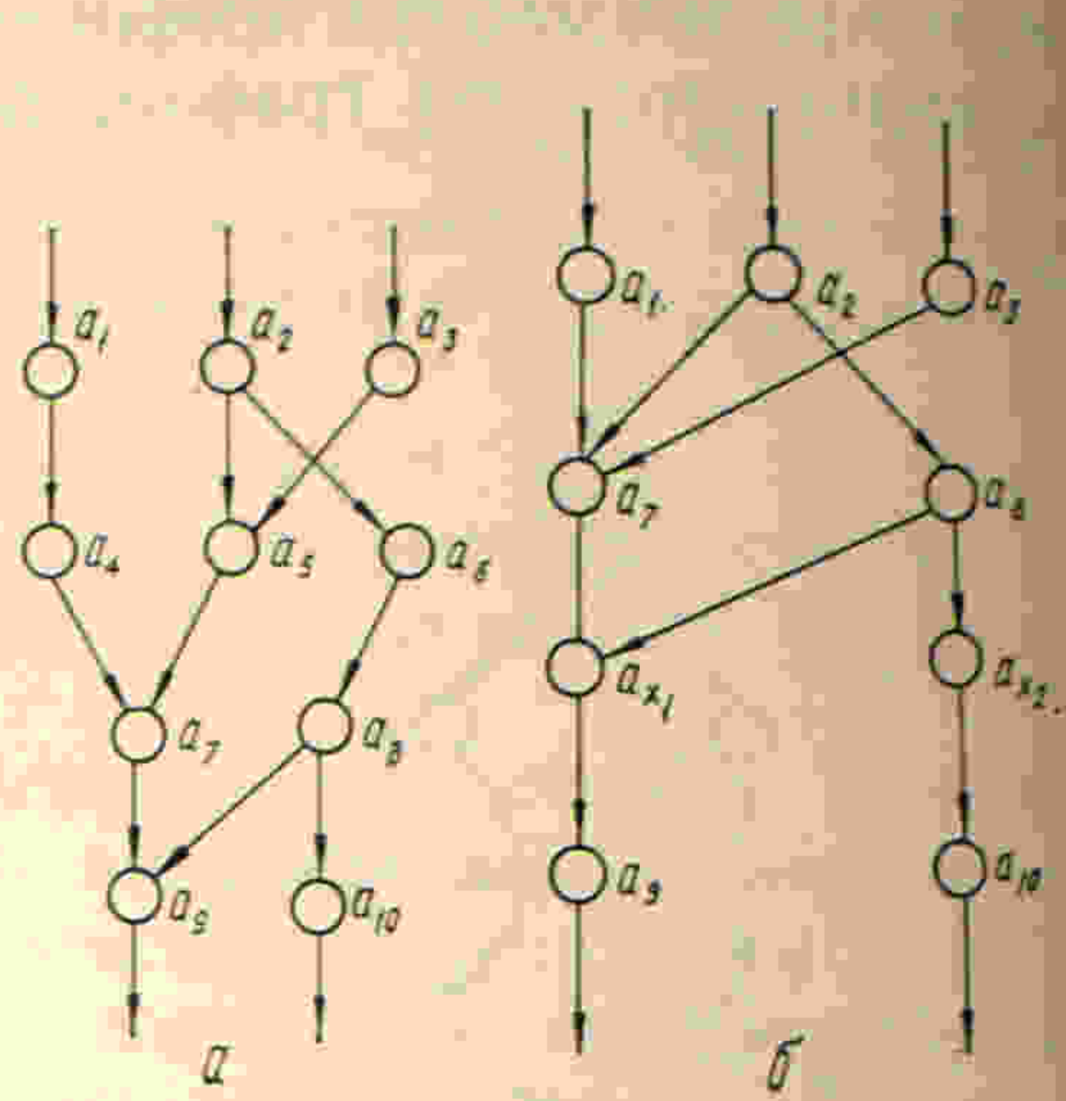


Рис. 15

(число операторов перехода после выполнения рассматриваемой процедуры над участком (а) сократилось на один):

```
   1 IF (B GT. 0.) GO TO 1
   2 A 1 = C (1)
   3 A 2 = C (2)
   4 GO TO 2
5  1 E 1 = R 1 * 3.
   6 E 2 = R 2 * 2.
   7 GO TO 3
8  2 A 3 = C (3)
   9 A 4 = C (4)
10 3 CONTINUE
```

```
б) 1 IF (B. GT. 0) GO TO 1
   2 A 1 = C (1)
   3 A 2 = C (2)
   8 A 3 = C (3)
   9 A 4 = C (4)
   7 GO TO 3
5  1 E 1 = R 1 * 3.
   6 E 2 = R 2 * 2.
10 3 CONTINUE
```

В результате выполнения процедуры переупорядочения линейных участков возникает не только прямой оптимизационный эффект, обусловленный удалением из программы операторов безусловного перехода, но и косвенный оптимизационный эффект, связанный с упрощением процесса построения и анализа управляющего и информационного графов программы, что облегчает выполнение других процедур и в ряде случаев позволяет увеличить их оптимизационный эффект. Например, за счет увеличения длины линейных участков в общем случае увеличивается оптимизационный эффект процедур, выполняемых на этих участках.

Локализация вычислений. Каждому набору начальных значений, как правило, соответствует свой путь выполнения программы, т. е. определенный набор и последовательность выполняемых операторов. При этом выполнение некоторых операторов для конкретного набора данных может быть функционально излишним, хотя эти операторы не являются излишними, так как для другого набора начальных значений их выполнение необходимо для сохранения смысла программы. Кроме того, при построении любой программы, вполне естественно (хотя бы для того, чтобы освободить ячейки, занимаемые переменными-операндами) вычислять выражение сразу же после того, как входящие в него переменные-операнды получат нужные значения. Смысл процедуры локализации вычислений заключается в преобразовании программы таким образом, чтобы для каждого набора начальных значений выполнялись только самые необходимые вычисления и значения переменных хранились в памяти минимальное время.

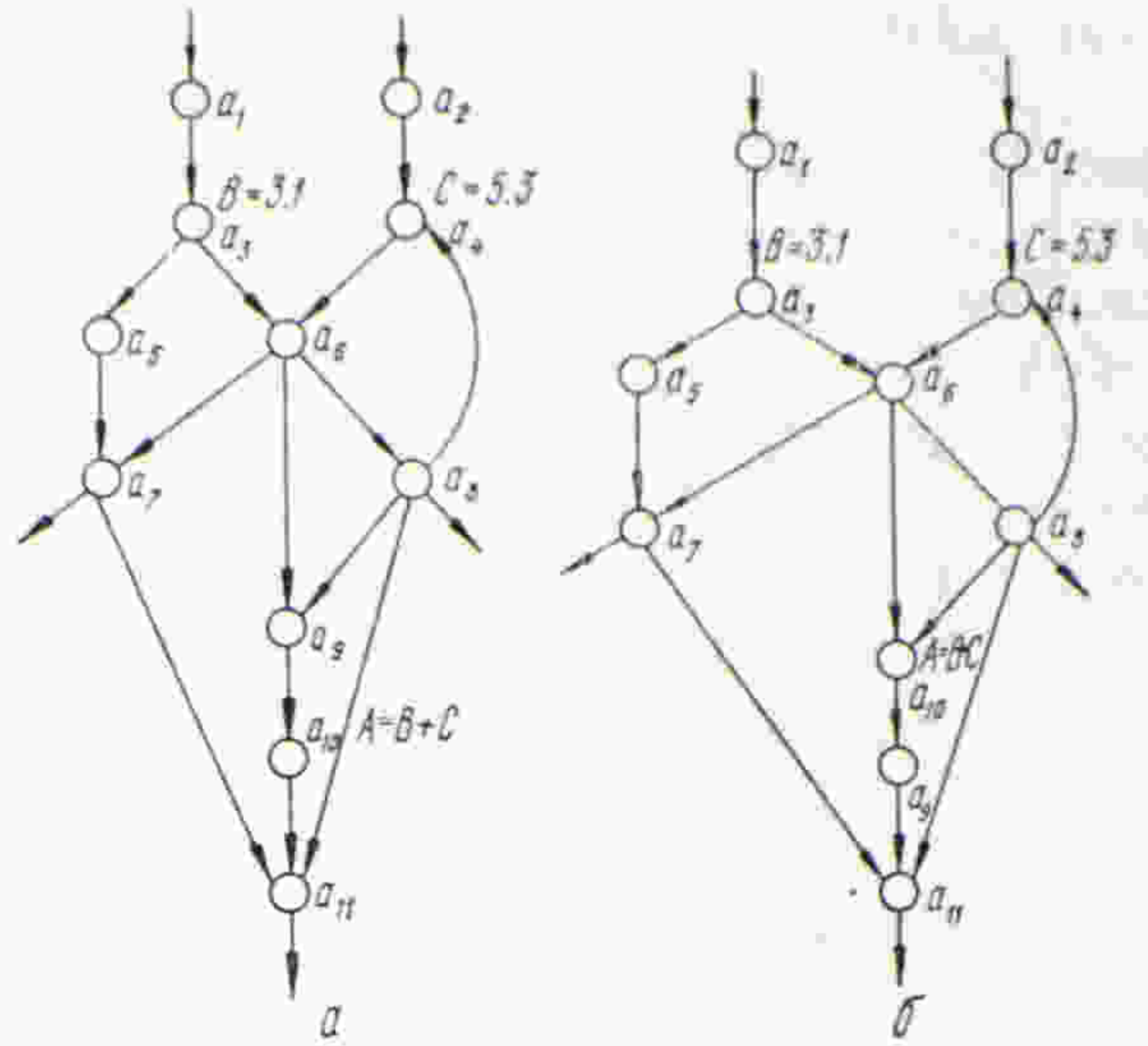


Рис. 16

Основные преобразования, выполняемые рассматриваемой процедурой, заключаются в перемещении отдельных операторов по направлению к начальной или конечной вершине графа программы. Направление перемещения операторов тесно связано с критерием оптимизации. Действительно, например, перемещение оператора присваивания к начальной вершине графа программы позволит сэкономить память, вследствие освобождения ячеек, занимаемых значениями переменных-операндов его правой части (после вычисления оператора присваивания при выполнении некоторых условий может исчезнуть необходимость в хранении в памяти значений переменных-операндов правой части этого оператора), перемещение оператора присваивания по направлению к конечной вершине графа программы позволит сократить время выполнения программы для тех наборов данных, для которых выполнение перемещаемого оператора является функционально излишним. Следует отметить, что при перемещении операторов по направлению к конечной вершине графа возможно также и сокращение времени хранения переменных-операндов перемещаемого оператора в памяти ЭВМ. Возможности выполняемых перемещений определяются выполнением ряда условий, в основе

которых лежат соотношения, учитывающие управляющие и информационные связи программы. Невыполнение этих условий, а следовательно, отсутствие возможности переместить оператор, указывает на то, что данный оператор принадлежит пути, на котором его выполнение является необходимым для сохранения смысла программы.

В частном случае для гамаков условия, определяющие возможность перемещения оператора по направлению к конечной вершине графа, сформулированы в процедуре втягивание в гамак.

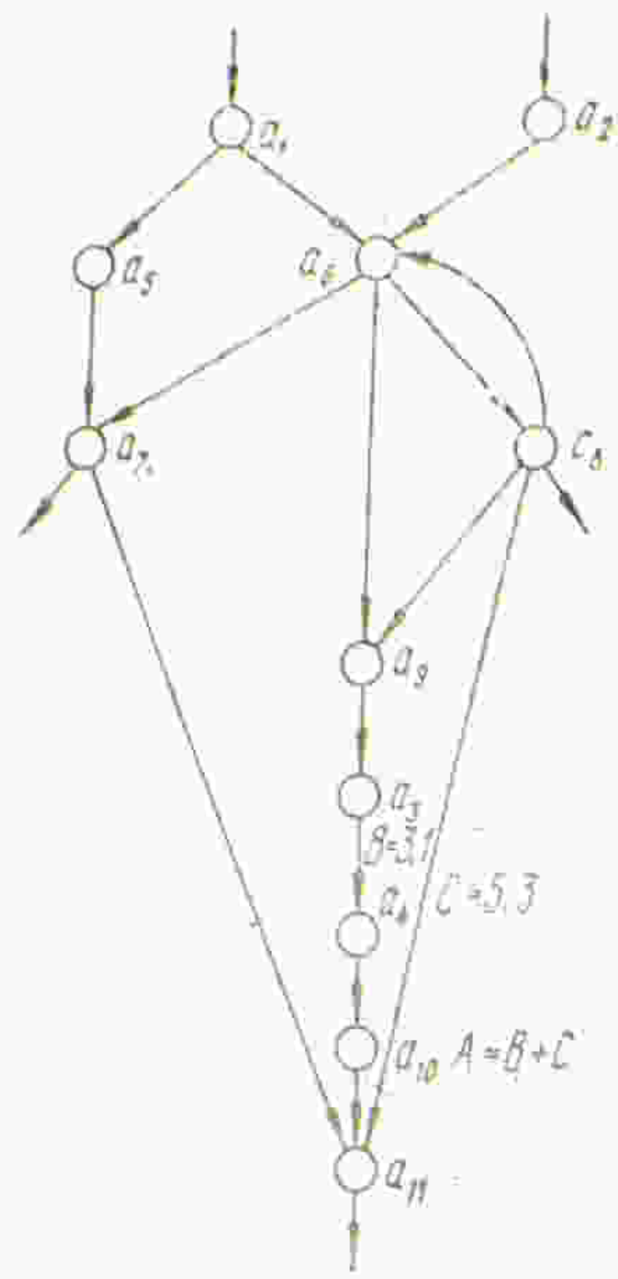


Рис. 17

Граф, показанный на рис. 16, *б*, является результатом выполнения процедуры локализации вычислений над графом, показанным на рис. 16, *а*. Здесь оператор a_3 определяет переменной B значение 3.1, а оператор a_4 — переменной C значение 5.3. Эти значения используются только оператором a_{10}, имеющим вид $A = B + C$. В результате выполнения процедуры локализации вычислений оператор a_{10} перемещается по направлению к начальной вершине графа программы и значение переменных B и C хранится в ячейке минимальное время. Заметим, что минимизации времени хранения значений переменных B и C в ячейках памяти можно добиться выполнением других преобразований, а именно (при выполнении некоторых условий): перемещением операторов a_3 и a_4 по направлению к оператору a_{10}. На рис. 17 показан граф, являющийся результатом выполнения этого преобразования над графом, показанным на рис. 16, *а*. Результатом выполненного преобразования является сокращение времени выполнения программы для некоторых наборов начальных данных и сокращение времени хранения значений переменных B и C в памяти ЭВМ. На приведенных рисунках хорошо видна зависимость действий, выполняемых рассматриваемой процедурой, от критериев оптимизации программы.

Оптимизационный эффект, получаемый в результате выполнения процедуры локализации вычислений, можно разделить на прямой и косвенный. Прямой оптимизационный эффект заключается в том, что после выполнения процедуры локализации вычислений преобразованная программа для некоторых наборов начальных значений будет выполняться быстрее, и сократится время хранения значений некоторых переменных в памяти ЭВМ. Косвенный оптимизационный эффект заключается в том, что сокращение времени хранения информации в памяти ЭВМ позволяет повысить эффективность выполнения других оптимизацион-

ных процедур, связанных с экономией памяти, например процедур глобальной и локальной экономии памяти [12].

Ликвидация константных распознавателей. Под распознавателем здесь понимается оператор условного перехода либо семантически аналогичный ему оператор, осуществляющий выбор следующего выполняемого оператора, т. е. приемника оператора условного перехода. Смысл процедуры ликвидации константных распознавателей заключается в уменьшении (быть может до одного) числа приемников операторов условного перехода. Это возможно в двух случаях: некоторые приемники совпадают между собой; область изменения условия (выбирающего выражения) ограничена, либо условие является константой.

Если у распознавателя имеется только один приемник или выбирающее выражение является константой, то такой распознаватель удаляется из программы, и на его место в программе помещается конструкция, указывающая на этот приемник (такой конструкцией, как правило, является оператор безусловного перехода). Если можно уменьшить, но нельзя сократить число приемников рассматриваемого распознавателя до одного, то этот распознаватель заменяется другим, более эффективным, для меньшего числа приемников. Например, участок (б) является результатом выполнения процедуры ликвидации константных распознавателей над участком (а), а участок (г) — результатом последовательного выполнения процедур подстановки [12] и выполнения константных распознавателей над участком (в):

```
а) IF (A) 2, 2, 3          б) IF (A. GT. 0) GO TO 3
   2 A = B + C                2 A = B + C
                                 .
                                 .
                              3 D = E * 3
   3 D = E * 3
в) A =. TRUE.              г) A =. TRUE.
      .                          .
      .                          .
   IF (A) D = B + C           D = B + C
   E = 3.                     E = 3.
```

Следует отметить, что процедура ликвидации константных распознавателей является повторной по отношению к ряду процедур, например к процедурам переупорядочения линейных участков; уменьшения числа слияний (см. гл. 2.3); устранения лишних меток (см. гл. 2.9). В частности, если на оператор с меткой 2 участка (б) не существует передач управления от остальных операторов программы, то над участком (б) может быть выполнена процедура устранения лишних меток, которая не может быть выполнена над участком (а). Для повышения оптимизационного эффекта рассматриваемой процедуры предварительно следует выполнить процедуры подстановки [12] и выполнения константных действий [8].

Использование эффективных распознавателей. Суть этой процедуры основана на том принципе, что использование нескольких простых логических выражений в распознавателе является более эффективным, чем использование одного эквивалентного им сложного логического выражения. Эта эффективность обусловлена тем, что логические выражения могут содержать подвыражения, вычисление которых не является необходимым для правильного функционирования программы. Например, логическое выражение (язык ПЛ/1) (A1 > B1) | (C1 < D1) содержит подвыражения A1 > B1 и C1 < D1, причем вычисление второго из них является лишним, если значение первого — «истина». Аналогично логическое выражение (A2 < B2) & (C2 > D2) содер-

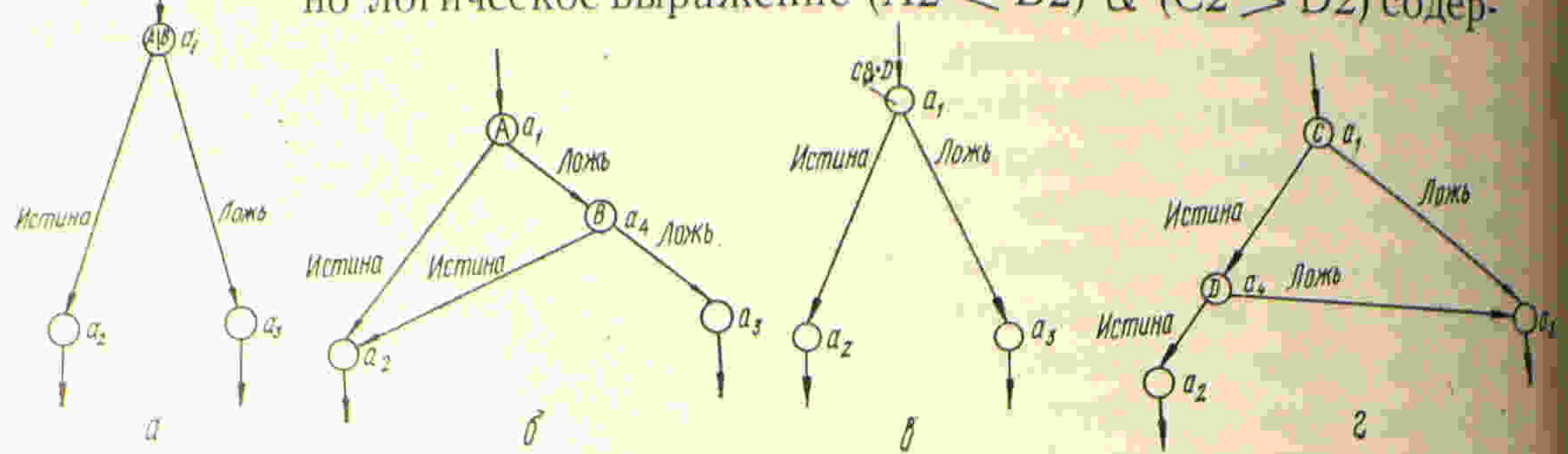


Рис. 18

жит подвыражения A2 < B2 и C2 > D2, причем здесь также вычисление второго из них является лишним, если значение первого — «ложь». Преобразования, выполняемые рассматриваемой процедурой, заключаются в декомпозиции логических выражений в распознавателях (для языков программирования высокого уровня в операторах условного перехода IF) на простые подвыражения.

Графы, показанные на рис. 18, *б* и *г*, являются результатами выполнения процедуры использование эффективных распознавателей над графами, показанными на рис. 18, *а* и *в* соответственно. Здесь через A, B, C, D обозначены логические выражения, которые, в частности, могут быть простыми логическими переменными; через a_4 обозначен дополнительный оператор, появляющийся в программе в результате выполнения рассматриваемой процедуры, реализующий один из двух более простых, чем исходный, распознавателей.

Например, участки (б) и (г) являются результатом выполнения рассматриваемой процедуры над участками (а) и (в) соответственно (язык ПЛ/1):

а) IF (A 1 > B 1) | (C 1 < D 1) THEN P = T;
б) IF A 1 > B 1 THEN P = T ELSE IF C 1 < D 1 THEN P = T;
в) IF (A 2 < B 2) & (C 2 > D 2) THEN E = F;
г) IF A 2 < B 2 THEN IF C 2 > D2 THEN E = F;

Заметим, что если через A, B, C, D обозначить логические выражения A1 > B1, C1 < D1, A2 < B2, C2 > D2 соответствен-

но, то участкам (а), (б), (в), (г) в некотором смысле будут соответствовать графы, показанные на рис. 18, *а*, *б*, *в*, *г* соответственно.

К действиям, выполняемым процедурой использования эффективных распознавателей, относятся также преобразования, обусловленные особенностями организации ветвлений в программах, написанных на языках высокого уровня. Например, участок (д) (язык ПЛ/1) в результате выполнения рассматриваемой процедуры преобразуется в участок (е), а участок (з)— в участок (ж):

```
д) IF A <= 3 THEN;
   ELSE B = C;
з) IF B > C THEN X = 0;
   ELSE GO TO L;
```

```
е) IF A > 3 THEN B = C
ж) IF B <= C THEN GO TO L;
   X = 0;
```

Объединение операторов условного перехода. Суть этой процедуры основана на том принципе, что использование одного сложного оператора условного перехода является более эффективным, чем использование эквивалентной ему совокупности простых операторов условного перехода. Преобразования, выполняемые рассматриваемой процедурой, заключаются в замене нескольких последовательных операторов условного перехода одним эквивалентным им оператором. Например, участок (б) является результатом выполнения процедуры объединения операторов условного перехода над участком (а) (язык КОБОЛ):

```
а) IF (D = 1) THEN MOVE A1 TO A2.
   IF (D = 2) THEN MOVE B1 TO B2.
   IF (D = 3) THEN MOVE C1 TO C2.
б) IF (D = 1) THEN MOVE A1 TO A2
   ELSE IF (D = 2) THEN MOVE B1 TO B2
   ELSE IF (D = 3) THEN MOVE C1 TO C2.
```

Уменьшение числа слияний. Суть этой процедуры состоит в сокращении числа слияний различных ветвей программы, т. е. удалении содержательно лишних передач управления на некоторые метки. Передача управления является лишней, если она выполняется на оператор безусловного перехода. Рассматриваемая процедура осуществляет поиск таких передач управления и устраняет их, заменяя метки в соответствующих операторах перехода. Если все передачи управления на некоторый оператор безусловного перехода являются содержательно лишними, то такой оператор устраняется из программы. Например, участок (б) является результатом выполнения процедуры уменьшения числа слияний над участком (а) (предполагается, что на оператор GO TO 3 осуществляется в программе только одна передача управления):

```
а) IF (A) 1, 2, 3        б) IF (A) 3, 2, 3
      ⋮                        ⋮
```

```
1 GO TO 3                    2 F = D ** E 1
2 E = D ** E 1               3 A = B
3 A = B
```

Рассматриваемая процедура также выполняет удаление операторов безусловного перехода, осуществляющих передачу управления на следующий оператор.

5. УСТРАНЕНИЕ ИЗБЫТОЧНЫХ ВЫЧИСЛЕНИЙ

Опишем процедуры, в результате выполнения которых может сократиться число производимых во время счета программы различных вычислений и преобразований данных. Устраняемые вычисления являются по существу избыточными, их удаление не меняет смысла программы. Такие избыточные вычисления появляются в программе по следующим причинам: незнанию программистом тонкостей реализации языка программирования; сложности программы и, вследствие этого, невозможности полного ее контроля; невнимательности программиста; вследствие выполнения различных преобразований программы (в процессе отладки, оптимизации и т. д.).

Подстановка. В большинстве программ, написанных на языках высокого уровня, существуют переменные, индексы и части арифметических выражений, которые во время трансляции можно заменить их числовыми значениями. Процедура, выполняющая такие преобразования, называется подстановкой:

```
а) A = 5.          б) D = SIN (X) + 10. 0 * D — 5.
   C = 10. 0
   .
   .
   D = SIN (X) + C * D — A
```

В результате выполнения процедуры подстановки участок (а) преобразуется в участок (б). Аналогично осуществляется подстановка логических переменных в выражениях и их частях. В результате выполнения процедуры подстановки сокращаются время выполнения программы и объем занимаемой ею памяти.

Отождествление переменных. Эта процедура в некотором смысле аналогична процедуре подстановки. Если в программе имеется оператор вида A = B, суть процедуры отождествление переменных заключается в замене переменной A переменной B или числовым значением переменной B во всех последующих операторах. Возможность выполнения такой замены определяется рядом условий. В зависимости от полноты их выполнения возможны четыре случая:

1. Переменная B при выполнении оператора A = B всегда, вне зависимости от набора начальных значений, имеет одно и то же числовое значение β. Оператор A = B удаляется из программы, а переменная A в ряде операторов заменяется значением β.

2. Переменная B при выполнении оператора A = B всегда, вне зависимости от набора начальных значений, имеет одно и то же числовое значение β. Оператор A = B остается в программе, а переменная A в некоторых операторах заменяется значением β.

3. Переменная B при выполнении оператора A = B может иметь различные значения, зависящие от набора начальных значений. Оператор A = B удаляется из программы, а переменная A в ряде операторов может заменяться переменной B.

4. Переменная B при выполнении оператора A = B может иметь различные значения, зависящие от набора начальных значений, а переменная A является индексируемой переменной. Оператор A = B остается в программе, а переменная A в некоторых операторах заменяется переменной B.

Следует отметить, что в результате выполнения процедуры отождествления переменных в первом и третьем случаях возникает прямой оптимизационный эффект, обусловленный удалением из программы оператора A = B, причем этот эффект существенно возрастает, если переменная A является индексируемой переменной. Прямой оптимизационный эффект возникает также и в четвертом случае, так как уменьшается число обращений к индексируемым переменным. Во втором случае в результате выполнения рассматриваемой процедуры возникают косвенный оптимизационный эффект (если переменная A является переменной с индексами, то возникает также и прямой оптимизационный эффект), заключающийся в том, что выполняемые преобразования позволяют повышать оптимизационный эффект других процедур, например, процедур выполнения константных действий и упорядочения типов данных [8]. В первом случае также (наряду с прямым оптимизационным эффектом) возникает косвенный оптимизационный эффект.

Удаление излишних операторов присваивания. Одним из наиболее трудно решаемых вопросов, возникающих при оптимизации программ, является уменьшение числа пересылок информации из ОЗУ в сумматор и обратно. Оптимальное решение в каждом конкретном случае можно найти при анализе программы, оттраслированной в машинные коды, однако некоторого улучшения в вышесказанном смысле можно достигнуть преобразованием программы в исходном языке с помощью процедуры удаления излишних операторов присваивания. Рассматриваемая процедура состоит в удалении из программы некоторых операторов присваивания и замене в некоторых других операторах (присваивания или условного перехода) переменных, являющихся

левой частью удаляемых операторов, выражениями, являющимися правой частью удаляемых операторов, т. е. выполняется объединение операторов. Например, на участке (б) показан оператор, являющийся результатом выполнения процедуры удаления излишних операторов присваивания над операторами участка (а):

```
а) C 1 = 5. + B 1 * D * * 2
   ⋮
   C 2 = B + SIN (R)
   ⋮
   C 3 = 5. + A (1) * A (2)
   C = (C 1 + C 2) * C 3
б) C = (5. + B 1 * D ** 2 + B + SIN (R)) * (5. + A (1) * A (2))
```

Оптимизационный эффект от выполнения рассматриваемой процедуры возникает вследствие того, что объединение мелких арифметических операторов в более крупные сокращает число пересылок информации из ОЗУ в сумматор и обратно и, следовательно, время выполнения программы. При замене только в одном операторе присваивания переменных, являющихся левой частью удаляемых операторов, выражениями, являющимися правой частью удаляемых операторов, всегда возникает положительный оптимизационный эффект, так как в этом случае просто происходит сокращение числа операций пересылок информации. При этом предполагается, что переменные, являющиеся левой частью удаляемых операторов, в других операторах не используются. Так как удаляемые операторы могут находиться в любом месте программы, то выполнение процедуры удаления лишних операторов присваивания требует анализа управляющего и информационного графов всей программы.

Выполнение константных действий. Суть этой процедуры состоит в предварительном (до выполнения программы) вычислении или быть может частичном вычислении (т. е. вычислении подвыражений) арифметических выражений, являющихся правой частью операторов присваивания или составной частью арифметических операторов условного перехода, и замене этих выражений (подвыражений) вычисленными значениями. Выполнение таких вычислений возможно в случае, когда все или некоторые аргументы арифметических выражений (подвыражений) являются числовыми значениями. Например, оператор A = 5. + + B * 3. + 4.7 в результате выполнения рассматриваемой процедуры преобразуется в оператор A = 9.7 + B * 3.

Арифметические выражения, у которых все или некоторые аргументы — числовые значения, могут находиться в исходной программе либо появляться после выполнения некоторых оптимизационных процедур, например процедуры подстановки или процедуры удаления излишних операторов присваивания. Ана-

логично константные действия выполняются в логических выражениях (подвыражениях). Положительный оптимизационный эффект возникает за счет того, что несколько аргументов и операции над ними заменяются одним значением.

Использование конструкции DATA. При объявлении имени переменной в языках программирования высокого уровня, как правило, можно также задать начальное значение этой переменной, которое будет присвоено переменной в момент загрузки объектной программы в память. В языке ФОРТРАН для этого используется конструкция DATA, в языке ПЛ/1 — конструкция INITIAL. Особенность применения конструкции DATA заключается в том, что присвоение с ее помощью начального значения переменной обеспечивает ввод этого значения, но не требует объема памяти для размещения каких-либо команд в объектной программе и времени для выполнения этих команд. Суть рассматриваемой процедуры состоит в устранении из программы всех возможных операторов присваивания, определяющих переменные числовые значения, и присвоении значений этим переменным с помощью конструкции DATA. Применение этой конструкции для присваивания значений переменным экономит как время выполнения программы, так и объем занимаемой ею памяти. Использовать конструкцию DATA вместо операторов присваивания следует осторожно, так как для многократно вызываемых подпрограмм применение рассматриваемой процедуры может изменить смысл программы. Следует заметить, что применение рассматриваемой процедуры возможно не только к программам, написанным на языках высокого уровня, но и к программам, написанным на языках более низкого уровня (в частности, на языке ассемблера), которые содержат конструкции, аналогичные конструкции DATA в ФОРТРАНе.

Упорядочение типов данных. Современные языки высокого уровня позволяют пользователям применять в одном арифметическом выражении величины различных типов, не заботясь об их взаимных преобразованиях. Преобразование одного типа величины в другой там, где это необходимо, обеспечивается транслятором одним из двух способов в процессе компиляции объектной программы: порождением специальных объектных кодов или вызовом соответствующих подпрограмм. В обоих случаях на выполнение соответствующих команд при выполнении программы затрачивается время. Процедура упорядочения типов данных преобразует программу, в результате чего число взаимных превращений одного типа величин в другой сводится к минимуму. Это достигается несколькими путями.

Тип результата выполнения арифметического выражения, стоящего в правой части оператора присваивания, преобразуется, как правило, в тип переменной, находящейся в левой части этого же оператора. Поэтому при выполнении процедуры упоря-

дочения типов данных все числовые константы арифметического выражения правой части оператора присваивания приводятся к типу переменной, являющейся левой частью этого оператора. Кроме того, показатель степени, если это возможно, преобразуется к целому типу. Например, оператор A = 5 + RM * 3 + + B преобразуется в A = 5. + RM * 3. + B (здесь предполагается, что переменная A типа REAL), а подвыражения вида A ** 8.0 и A ** B, где B — переменная типа REAL, принимающая целые значения, преобразуются в подвыражения A ** 8 и A ** I соответственно, где I переменная типа INTEGER и значение переменной I равно значению переменной B.

Таким же образом осуществляются преобразования, устанавливающие соответствие между типами переменных правых и левых частей арифметических выражений.

Рассмотрим следующий пример:

```
A = I * X + (I + 3) * Y + I * 2.
```

При вычислении этого выражения, как правило, будет сделано два преобразования, а в некоторых трансляторах даже три — из типа INTEGER в тип REAL. В результате выполнения процедуры упорядочения типов данных рассматриваемый оператор будет преобразован к виду

```
V = I
A = V * X + (V + 3.) * Y + V * 2
```

Аналогично группа операторов участка (б) (являющегося результатом выполнения рассматриваемой процедуры над участком (а)) выполняется быстрее операторов участка (а):

```
а) A = B * I          б) AI = I
   C = (D + I) * 3. 5     A = B * AI
   E = D * F/I            C = (D + AI) * 3.5
                          E = D * F/AI
```

Полученное сокращение времени выполнения операторов достигается за счет уменьшения числа преобразований переменных из типа INTEGER в тип REAL.

Даже самые простые конструкции могут быть очень неэффективными вследствие неправильной организации преобразований типов данных:

```
      DO 1 I = 1, 100
    1 X (I) = I
```

При выполнении этого цикла будет сделано 100 преобразований переменных из типа INTEGER в тип REAL. Можно запись рассматриваемого цикла заменить другой, которая хотя и более длинная, но обеспечивает более быстрое его выполнение и для большинства трансляторов требует меньше памяти, так как в получаемых после трансляции объектных кодах отсутствуют команды, обеспечивающие преобразования типов данных:

```
      A = 0. 0
      DO 1 I = 1, 100
      A = A + 1
    1 X (I) = A
```

Если арифметическое выражение используется не в операторе присваивания, а в операторе условного перехода, то выполнение рассматриваемой процедуры также обеспечивает уменьшение суммарного числа преобразований типов данных. Например, оператор IF (A. GE. 10) GOTO5 преобразуется в оператор TF (A.GE.10.) GOTO5

Как правило, суммы нескольких слагаемых при отсутствии скобок вычисляются последовательно, начиная с первого. Объединение слагаемых одного типа в группы сокращает число лишних преобразований типов данных. Заключение таких групп в скобки также уменьшает для некоторых трансляторов время выполнения преобразуемых операторов. Например, в выражениях

$$R = I1 + R1 + DP1 - I2 - R2 - DP2$$
$$R = I1 - I2 + R1 - R2 + DP1 - DP2$$
$$R = (I1 - I2) + (R1 - R2) + (DP1 - DP2),$$

где переменные I1, I2 — типа INTEGER; R, R1, R2 — типа REAL и DP1, DP2 — типа DOUBLE PRECISION, рассматриваемые операторы присваивания с алгебраически эквивалентными правыми частями расположены в порядке уменьшения времени их выполнения. Аналогично при выполнении операции умножения необходимо упорядочивать сомножители по их типам. Следует отметить, что, как правило, преобразование данных из типа INTEGER в тип REAL являются более «дорогим» в смысле ресурсов ЭВМ, чем преобразования данных из типа REAL в тип DOUBLE PRECISION. Как показывает анализ, число лишних преобразований типов данных в программах велико, что определяет заметную эффективность рассматриваемой процедуры.

Оптимизация выражений. Суть процедуры оптимизации выражений состоит в поиске в программе семантически эквивалентных подвыражений и размещении этих подвыражений в программе таким образом, чтобы можно было вычеркнуть из программы хотя бы одно из этих подвыражений. Подвыражения считаются семантически эквивалентными, если они алгебраически эквивалентны (т. е. их можно преобразовать друг в друга, применив алгебраические преобразования), и их соответствующие операнды имеют одинаковые числовые значения. Согласно этому определению, задача поиска семантически эквивалентных подвыражений состоит из двух подзадач: поиска в программе подвыражений, использующих операнды, имеющие одинаковые числовые значения; поиска в программе алгебраически эквивалентных подвыражений.

Первая подзадача решается на основании анализа управляющего и информационного графов программы [3], а вторая — на основании применения алгоритмов преобразования арифметических выражений к некоторому стандартному виду [11]. Эти алгоритмы основаны на понижении ранга арифметических выражений до двух, а затем объединении простых арифметических

выражений в укрупненные и упорядочении операндов укрупненных выражений по приоритетам. На практике обе подзадачи решаются одновременно, что позволяет более эффективно находить искомые подвыражения.

Задача размещения в программе найденных семантически эквивалентных подвыражений решается на основе анализа управляющего и информационного графов программы. Например, операторы участка (б) являются результатом выполнения рассматриваемой процедуры над операторами участка (а):

```
а) C1 = D + (A — 5.)/B        б) Y 1 = (— A + 5.)/B
   ⋮                              C1 = D — Y1
   C2 = SIN (X) + (5. — A)/B      ⋮
                                  C2 = SIN (X) + Y1
```

В программу введена дополнительная переменная Y1, как правило, генерируемая системой оптимизации. Если программа содержит операторы присваивания, правые части которых, являющиеся арифметическими выражениями, семантически эквивалентны, то в этом случае можно обойтись без генерации дополнительной переменной (такой контекст, в котором можно обойтись без генерации дополнительной переменной, можно создавать путем предварительного применения вспомогательной процедуры переупорядочения операторов присваивания (см. гл. 2.9)). Например, участок (г), являющийся результатом выполнения процедуры оптимизации выражений над участком (в), не содержит такой переменной:

```
в) X1 = A1 * B1        г) X1 = A1 * B1
   ⋮                      ⋮
   X2 = A1 * B1           X2 = X1
```

Здесь предполагается, что между операторами рассмотренных фрагментов не содержится переопределений переменной X1.

Процедура оптимизация выражений выполняется не только в пределах области оптимизации, состоящей из всей программы или нескольких операторов, но и в пределах одного оператора. Например, участок (е) является результатом выполнения процедуры оптимизация выражений над участком (д):

```
д) E1 = (A + B) * C + D * * (A + B)   е) Y1 = A + B
                                          E1 = Y1 * C + D * * Y1
```

Выполнение процедуры оптимизации выражений в пределах одного оператора включает в себя также выделение общих множителей в арифметических выражениях, которое приводит к сокращению числа объектных команд, необходимых для реализации рассматриваемых выражений. Например, участок (ж) является результатом выполнения процедуры оптимизации выражений над участком (з) (в результате выполнения этой процедуры в пра-

вой части оператора участка (з) произведено выделение обоих множителей):

з) X = 2. * Y + (A — 1) * P + 2. * T
ж) X = 2. * (Y + T) + (A — 1) * P

При использовании процедуры оптимизации выражений следует помнить, что преобразования, выполняемые этой процедурой могут привести к изменению результата исполнения операторов. Это связано со спецификой выполнения операций умножения и деления с переменными типа INTEGER.

Процедура оптимизации выражений может выполняться на уровне входных языков (таких как ПЛ/1, АЛГОЛ, ФОРТРАН) или на уровне промежуточных языков, где арифметические выражения представляются в виде бинарных операций. Оптимизационный эффект процедуры оптимизации выражений заключается в сокращении времени выполнения программы и уменьшении объема занимаемой ею памяти.

Устранение неиспользованных определений. По разным причинам в программах пользователей встречаются случаи, когда переменная, которой присвоено значение, в дальнейшем нигде не используется, или же между двумя определениями этой переменной нет ни одного ее использования. Такие определения переменных назовем неиспользованными. Оптимизационная процедура устранения неиспользованных определений удаляет из программы лишние присваивания значений переменным, например:

```
11   A = SIN (X)
      :
12   C = 18. + E
      :
13   D = A — C
```

Если переменная D не используется на участке программы за оператором с меткой 13 (предполагается, что из этой области нет передач управления в область, предшествующую оператору с меткой 13), то устранение оператора D = A — C не повлияет на результат выполнения программы. После выполнения этой процедуры над оператором присваивания D = A — C может оказаться, что определения переменных, находящихся в правой части устраненного оператора, также являются неиспользованными. Если для переменных A и C выполняются соответствующие условия, их определения (операторы с метками 11 и 12) могут быть удалены из программы. Если переменная описана в конструкции COMMON, а также если она является выходным параметром подпрограммы или подпрограммы-функции, то последним определением таких переменных является последний оператор соответствующего модуля. При выполнении рассматриваемой процедуры над текстом только одного модуля следует

помнить, что каждый вызов подпрограммы в тексте преобразуемого модуля является потенциальным использованием и/или определением всех переменных, описанных в конструкции COMMON этого модуля. На основе анализа информационного графа программы можно формализовать понятие неиспользованного определения следующим образом. Разделим все операторы программы на два множества: используемых и неиспользуемых определений. Множество используемых определений строится в результате выполнения итерационного процесса, который определяется по индукции.

Базис индукции. На первой итерации в это множество включаются операторы вывода информации и операторы, присваивающие внешним переменным значения, соответствующие висячим (т. е. из которых не выходит ни одной стрелки) вершинам на информационном графе.

Шаг индукции. На $k+1$-й итерации рассматриваемое множество дополняется операторами, соответствующими на информационном графе вершинам, из которых выходят стрелки, входящие в вершины, соответствующие операторам, включенным на k-й итерации во множество используемых определений. После окончания итерационного процесса будет построено множество использованных определений. Операторы, не вошедшие в это множество, составляют множество неиспользованных определений.

Процедуру устранения неиспользованных определений следует выполнять одной из последних в последовательности оптимизационных процедур. Это обусловлено тем, что в процессе выполнения некоторых оптимизационных преобразований в программу для удобства сознательно могут быть введены неиспользованные определения. В результате выполнения процедуры устранения неиспользованных определений уменьшается время счета программы и объем занимаемой ею памяти.

Устранение идентичных операторов. Пусть имеется некоторое множество операторов присваивания $a_{k_1}, a_{k_2}, \ldots, a_{k_m}$, $m \geqslant 1$, определяющих переменной A некоторое значение, которое обозначим через α_0. Оператор $a_{\hat{k}}$ назовем идентичным операторам $a_{k_1}, a_{k_2}, \ldots, a_{k_m}$, если выполняются следующие условия: операторы $a_{k_1}, a_{k_2}, \ldots, a_{k_m}, a_{\hat{k}}$ являются семантически эквивалентными (т. е. выполняют одни и те же операции над одними и теми же переменными и присваивают переменной A одно и то же значение); из выполнения оператора $a_{\hat{k}}$ в процессе выполнения программы следует, что до него был выполнен хотя бы один оператор a_{k_r}, причем последовательность выполняемых операторов $a_{j_1}, a_{j_2}, \ldots, a_{j_s}$, $s \geqslant 2$, где $j_1 = k_r$, $j_s = \hat{k}$, $r \in (\overline{1, m})$, не содержит операторов a_{j_t}, $t \in (\overline{2, s-1})$, определяющих переменной A

значение $\alpha_1 \neq \alpha_0$. Рассматриваемая процедура удаляет из программы идентичные операторы. Участок (б) является результатом выполнения процедуры устранения идентичных операторов над участком (а):

```
а)  IF (C) 1, 2, 2
 1  D = 3. 5
    GOTO 3
      .
      .
      .
 2  D = 3. 5
      .
      .
      .
 3  D = 3. 5
    E = A + B**2
```

```
б)  IF (C) 1, 2, 2
 1  D = 3. 5
    GOTO 3
      .
      .
      .
 2  D = 3. 5
      .
      .
      .
 3  E = A + B**2
```

Участок программы между операторами с метками 2 и 3 является линейным и не содержит операторов, определяющих переменной D некоторых значений. Оператор с меткой 3 идентичен операторам с метками 1 и 2, так как выполняются перечисленные выше условия, и он удаляется из программы. Процедура устранения идентичных операторов сокращает время выполнения программы и объем занимаемой ею памяти.

Устранение невыполняемых операторов. В реальных программах могут встретиться участки, содержащие невыполняемые операторы, т. е. такие операторы, которые не выполняются при любых наборах начальных данных. Анализ реальных программ показывает, что появление таких операторов можно объяснить двумя причинами:

1) в процессе отладки программы или ее модификации программист либо забывает о таких операторах, либо не хочет их искать, либо просто не в состоянии проследить за всеми последствиями, вызванными исправлением некоторых участков программы;

2) большая логическая или информационная сложность программы не позволяет программисту «увидеть» такие операторы или даже целые невыполняемые участки программы.

Смысл рассматриваемой процедуры состоит в удалении из программы невыполняемых операторов. Участок (б) является результатом выполнения процедуры устранения невыполняемых операторов над участком (а):

```
а)  IF (A 1) 1, 2, 3
    A = B + C
 1  E 1 = 5.
      .
      .
      .
 2  E 2 = 6.
      .
      .
      .
 3  E 3 = 7.
```

```
б)  IF (A 1) 1, 2, 3
 1  E 1 = 5.
      .
      .
      .
 2  E 2 = 6.
      .
      .
      .
 3  E 3 = 7.
```

Это один из простых случаев появления невыполняемых операторов в программе. Такие операторы можно обнаружить в процессе элементарного семантического контроля, а именно, проверки наличия меток (и передачи на них управления) у операторов, следующих за операторами перехода условного или безусловного. Рассмотрим более сложный случай появления невыполняемых операторов в программе:

```
в)  IF (M) 1, 5, 2
    :
  1 IF (X) 3, 2, 2
    :
  3 IF (M) 5, 6, 6
  6 A = B + C
  5 E 1 = 5.
  2 E 2 = 6.
```

```
г)  IF (M) 1, 5, 2
    :
  1 IF (X) 5, 2, 2
    :
  5 E 1 = 5.
  2 E 2 = 6.
```

Появление на участке (в) невыполняемого оператора A = B + + C обусловлено логической и информационной сложностью программы. Действительно, оператор с меткой 1 выполняется только при отрицательных значениях переменной M, а следовательно, оператор с меткой 3 также выполняется только при отрицательных значениях переменной M (здесь предполагается, что участки программы между условными операторами перехода являются линейными, и на оператор с меткой 3 нет никаких других передач управления). Из этого следует, что оператор с меткой 3 никогда не передает управление на оператор с меткой 6, т. е. на оператор A = B + C. Участок (г) является результатом выполнения процедуры устранения невыполняемых операторов, а затем процедуры ликвидации константных распознавателей (которая удаляет оператор IF (M) 5, 6, 6) над участком (в). Следует заметить, что в общем случае поиск в программе невыполняемых операторов требует достаточно сложного анализа управляющего и информационного графов программы.

Сокращение числа индуктивных переменных. Обозначим через $a_1, a_2, \ldots, a_k$ последовательность выполняемых операторов программы, а через $\alpha_1, \alpha_2, \ldots, \alpha_m^m$ — значения, присваиваемые переменной A в этой последовательности. Если значения $\alpha_1, \alpha_2, \ldots, \alpha_m$ образуют арифметическую прогрессию, то переменная A называется индуктивной переменной, определенной на последовательности операторов $a_1, a_2, \ldots, a_k$. В частности, рекурсивно-определяемые переменные вида A = A + C где (C — константа), находящиеся в зоне, являются индуктивными переменными. При этом в качестве последовательности операторов $a_1, a_2, \ldots, a_k$ рассматривается полностью развернутая зона, т. е. последовательность копий тела зоны, каждая из которых соответствует одному ее выполнению. Важным свойством индуктивных переменных является их линейная связь друг с другом на последова-

тельностях операторов, где эти переменные определены. Это свойство индуктивных переменных лежит в основе процедуры сокращения числа индуктивных переменных, суть которой заключается в следующем. Если требуется значение индуктивной переменной только вне последовательности операторов a_1, a_2, ..., a_k, на которой она определена, то значение этой переменной, используя ее линейную связь с другими индуктивными переменными (предполагается, что такие переменные имеются), можно вычислять только один раз при выполнении последнего оператора последовательности a_1, a_2, ..., a_k. В частности, если значения всех индуктивных переменных, определенных на последовательности a_1, a_2, ... a_k, требуются только вне этой последовательности, то при выполнении операторов a_1, a_2, ..., a_k можно вычислять значение только одной индуктивной переменной, а значения остальных вычислять после выполнения оператора a_k. Относительно рекурсивно-определяемых переменных вида A = A + C, находящихся в зоне, значения которых используются только вне зоны, это означает, что значения этих переменных можно вычислять только один раз, после выхода из зоны. Например, участок (б) является результатом выполнения процедуры сокращения числа индуктивных переменных над участком (а):

```
а)  A = A 1
    B = B 1
    DO 1 I = 1,100
    X (I) = I * 3. 5
    A = A + C
    B = B + D
  1 CONTINUE
```

```
б)  A = A 1
    B = B 1
    DO 1 I = 1,100
    X (I) = I * 3. 5
  1 CONTINUE
    A = A 1 + C * 100
    B = B 1 + D * 100
```

Здесь A, B, I — индуктивные переменные, между которыми существуют следующие линейные зависимости:

```
A = A 1 + C * I
B = B 1 + D * I
```

Использование этих зависимостей позволило значительно упростить исходный участок (а).

Аналогично тому, как индуктивные переменные были определены относительно операции сложения, эти же переменные определяются относительно операции умножения. Пусть в последовательности выполняемых операторов a_1, a_2, ..., a_k значения, присваиваемые переменной A, образуют геометрическую прогрессию. Тогда переменную A будем также называть индуктивной переменной, определенной на последовательности операторов a_1, a_2, ..., a_k. В частности, рекурсивно-определяемые переменные вида A = A * C (где C — константа), находящиеся в зоне, являются индуктивными переменными. Для рассматриваемых индуктивных переменных на множестве операторов, где они

определены, также существует связь, аналогичная линейной. Выполнение процедуры сокращения числа индуктивных переменных, основанное на этой связи, позволяет сократить число вычислений индуктивных переменных. Например, участок (г) является результатом выполнения рассматриваемой процедуры над участком (в):

```
в) E = E 1
   F = F 1
   DO 1 I = 1,50
   Y (I) = I/7. 8 + 3. 6
   E = E * C
   F = F * D
 1 CONTINUE
```

```
г) E = E 1
   F = F 1
   DO 1 I = 1,50
   Y (I) = I/7. 8 + 3. 6
 1 CONTINUE
   E = E 1 ** 50 * C
   F = F 1 ** 50 * D
```

Здесь E, F и I — индуктивные переменные, между которыми существуют следующие зависимости:

```
E = E 1 ** I * C
F = F 1 ** I * D
```

Использование этих зависимостей позволило значительно упростить исходный участок (в).

Следует заметить, что процедура сокращения числа индуктивных переменных выполняет преобразования, в некотором смысле противоположные преобразованиям, выполняемым процедурой пребразования рекурсивно-определяемых переменных. Однако сочетание этих преобразований с выносом операций над рекурсивно-определяемыми переменными из зоны (цикла), несмотря на осуществляемую замену менее сложных операций более сложными, позволяет получать положительный оптимизационный эффект. Рассмотренная процедура замены проверок условия в некотором смысле является частным случаем процедуры сокращения числа индуктивных переменных, так как в ее основе лежит то же свойство линейной связи между переменными.

Выполнение аналитических преобразований. Суть этой процедуры основана на алгебраических формулах $a * 1 = a$; $1 * a = a$; $a * 0 = 0$; $0 * a = 0$; $a/a = 1$; $a/1 = a$; $0/a = 0$; $a ** 1 = a$; $0 + a = a$; $a + 0 = a$; $a - a = 0$ и т. д. Она сводится к замене арифметических выражений, соответствующих левым частям этих формул, выражениями, соответствующими правым частям этих формул.

Возможность выполнения аналитических преобразований возникает вследствие наличия соответствующих конструкций в исходной программе или в результате выполнения оптимизационных процедур подстановки, отождествления переменных, удаления излишних операторов присваивания, выполнения константных действий и др. Участок (б) является результатом последовательного применения процедур постановки, выполнения кон-

стантных действий, отождествления переменных и затем процедуры выполнения аналитических преобразований к участку (а):

```
а) C = 0. 7                б) A = B
   X = E                      D = F + 0. 5
   A = B ** (C + 0. 3)
   D = F + E * 0. 5/X
```

Оптимизационный эффект процедуры выполнения аналитических преобразований определяется сокращением числа выполнений арифметических операций.

Удаление тождественных операторов. Суть этой процедуры состоит в удалении из программы тождественных операторов, т. е. операторов вида A = A. Такие операторы могут появляться в программе в результате выполнения оптимизационных и других преобразований.

Устранение лишних скобок. Суть этой процедуры заключается в устранении пар скобок в арифметических или логических выражениях, являющихся содержательно излишними. При этом следует выделить два случая: в скобках заключено выражение, уже заключенное в скобки; скобки группируют аргументы арифметического или логического выражения определенным образом. В первом случае самые внешние скобки просто удаляются. Во втором случае следует учитывать специфику выполнения на ЭВМ арифметических операций над малыми и большими числами, когда возможна потеря точности, и числами типа INTEGER, когда от порядка аргументов может существенно зависеть результат выполнения таких операций. Поэтому процедуру устранения лишних скобок в таких случаях следует выполнять после анализа числовых значений и типов аргументов. Участок (б) является результатом выполнения процедуры устранения лишних скобок над участком (а):

```
а) A 1 = ((B + C)) * E          б) A 1 = (B + C) * E
   :                               :
   A 2 = ((B + C) + D + R) * F     A 2 = (B + C + D + R) * F
```

Оптимизационный эффект, возникающий в результате выполнения этой процедуры, обусловлен тем, что появление лишних скобок в ряде случаев приводит к появлению в объектной программе, полученной после трансляции, дополнительных команд пересылки информации из ОЗУ в сумматор и обратно.

Оптимизация использования быстрых регистров. Большинство ЭВМ имеют, как правило, небольшой набор быстрых регистров, которые отличаются от ячеек основной памяти тем, что выборка информации, находящейся в быстрых регистрах, требует значительно меньше времени, чем выборка из основной памяти. Поэтому желательно использовать регистры как можно более эффективно. Суть процедуры оптимизации использования быстрых

регистров заключается в определении и реализации наиболее эффективного способа размещения значений переменных программы на регистрах. При этом решаются две задачи: определения множества переменных, значения которых должны храниться на регистрах в каждый момент выполнения программы, и распределения таких переменных на множестве доступных регистров. Например, в быстрых регистрах размещают переменные, которые меняют свои значения наиболее часто, т. е. часто считываются и записываются в ячейки памяти. Это позволяет сократить время доступа к ним, и, следовательно, ускорить весь процесс обработки данных [48].

Рассмотрим выполнение процедуры оптимизации использования быстрых регистров при распределении на них параметров цикла (управляющей переменной, ее шага и граничного значения на регистрах в течение выполнения цикла). В результате выполнения рассматриваемой процедуры переменные I, K и L, являющиеся параметрами заголовка цикла (язык ПЛ/1)

DO I = A TO K BY L;

будут размещены на быстрых регистрах, что сократит время выполнения цикла.

Процедура оптимизации быстрых регистров выполняется, как правило, на уровне языков ассемблера, однако некоторые возможности улучшения распределения регистров имеются при выполнении ряда оптимизационных процедур, например, разделения циклов, сокращения числа вычислений индексных функций, локализации вычислений, удаления излишних операторов присваивания.

Переупорядочение подвыражений. Для большинства ЭВМ команды пересылки информации с сумматора в ОЗУ и обратно составляют значительную часть всех выполняемых команд программы [71]. Процедура переупорядочения подвыражений, выполняемая на уровне языка ассемблера, позволяет сократить число таких пересылок информации. Областью оптимизации рассматриваемой процедуры является группа команд, соответствующих исходному арифметическому выражению, записанному на уровне входного языка. Оптимизационный эффект, получаемый от выполнения этой процедуры, определяется, в основном, системой команд ЭВМ и алгоритмами трансляции арифметических выражений с языков высокого уровня в язык ассемблера. Большинство таких алгоритмов трансляции, как правило, базируется на польской (бесскобочной) записи выражений.

Результат выполнения процедуры переупорядочения подвыражений над группой команд языка ассемблера участка (а), полученного трансляцией выражения A * B — (C + D) * E с использованием польской (бесскобочной) записи, для ЭВМ с одним регистром показан на участке (б) (здесь предполагается, что система команд ЭВМ не содержит обратного вычитания):

```
а) ЗАГР    A          б) ЗАГР    C
   УМНЖ    B             СЛОЖ    D
   ЗАП     ЯЧЕЙКА 1      УМНЖ    E
```

ЗАГР	C	ЗАП	ЯЧЕЙКА 1
СЛОЖ	D	ЗАГР	A
УМН	E	УМНЖ	B
ЗАП	ЯЧЕЙКА 2	ВЫЧТ	ЯЧЕЙКА 1
ЗАГР	ЯЧЕЙКА 1		
ВЫЧТ	ЯЧЕЙКА 2		

Применение процедуры переупорядочения подвыражений к выражению A * B — (C + D) * E заключается в том, что подвыражение (C + D) * E вычисляется раньше подвыражения A * B. Это позволяет выполнить трансляцию так, что число команд языка ассемблера, соответствующих исходному выражению, сокращается с девяти до семи, а число используемых промежуточных ячеек сокращается с двух до одной.

Результат выполнения процедуры переупорядочения подвыражений над группой команд ассемблера участка (в), полученного трансляцией выражения A + B + C * D + E * F с использованием польской записи, для ЭВМ с двумя регистрами показан на участке (г) (здесь обозначение каждой команды содержит номер соответствующего регистра):

в)	ЗАГР 1	A	г) ЗАГР 1	C
	СЛОЖ 1	B	УМНЖ 1	D
	ЗАГР 2	C	ЗАГР 2	E
	УМНЖ 2	D	УМНЖ 2	F
	ЗАП 1	ЯЧЕЙКА	СЛОЖ 1	РЕГИСТР 2
	ЗАГР 1	E	ЗАГР 2	A
	УМНЖ 1	F	СЛОЖ 2	B
	СЛОЖ 1	РЕГИСТР 2	СЛОЖ 1	РЕГИСТР 2
	СЛОЖ 1	ЯЧЕЙКА		

Применение процедуры переупорядочения подвыражений к выражению A + B + C * D + E * F заключается в том, что подвыражение A + B вычисляется после вычисления подвыражений C * D и E * F. Это позволяет выполнить трансляцию так, что число команд ассемблера, соответствующих исходному выражению, сокращается с девяти до восьми. Аналогично рассматриваемая процедура выполняется для ЭВМ с тремя регистрами и более.

Для многорегистровых ЭВМ процедуру переупорядочения подвыражений в некоторых случаях можно выполнить на уровне входного языка путем переупорядочения подвыражений исходного выражения.

6. ЭФФЕКТИВНАЯ ОРГАНИЗАЦИЯ ПОДПРОГРАММ

Подпрограммы, подпрограммы-функции и процедуры — это средства создания модульных программ, активно используемые разработчиками программ большого объема. Стремление к созданию модульных программ объясняется следующими факторами.

1. Отдельные модули могут создаваться и отлаживаться различными разработчиками независимо друг от друга.

2. Модульные программы легко сопровождать и модифицировать. Исправление отдельного модуля вызывает минимальные изменения в связанных с ним по управлению и информации других модулях.

3. Модульные программы обладают повышенной надежностью, так как при их разработке существует возможность распределения работ по созданию модулей различной сложности и важности между программистами различной квалификации.

4. Можно создавать библиотеки наиболее употребительных подпрограмм.

5. Проще становится процедура загрузки в оперативную память всей программы.

6. Возникает много естественных контрольных точек для наблюдения за продвижением проекта.

Однако вместе с тем модульный подход требует дополнительных затрат ресурсов ЭВМ. Другими словами, применение средств модульного программирования эффективно только на этапах создания, отладки и модификации программ, а не на этапе их счета, когда выполнение команд, обеспечивающих вызов соответствующего модуля, является функционально излишним. Описываемые процедуры позволяют эффективно использовать имеющиеся в языках высокого уровня возможности написания подпрограмм и минимизировать при этом затрачиваемые ресурсы ЭВМ.

Сокращение числа вызовов подпрограмм. Связь подпрограмм с основной программой обеспечивается использованием операторов вызова в основной программе. Когда критерием оптимизации является время счета программы, отработка операторов вызова модулей, как правило, требующая выполнения нескольких или даже нескольких десятков машинных команд, является излишней.

Процедура сокращения числа вызовов подпрограмм осуществляет замену операторов, обеспечивающих вызов модулей, телами этих модулей и замену формальных параметров в теле этих модулей соответствующими им фактическими параметрами. При наличии в основной программе одного вызова модуля в результате выполнения этой процедуры сокращаются время выполнения программы и объем занимаемой ею памяти за счет устранения команд связи модулей.

Наличие в основной программе двух и более вызовов одного и того же модуля, помимо сокращения времени выполнения основной программы, влечет за собой, как правило, увеличение объема памяти, занимаемой основной программой. Исключением является тот случай, когда число команд, содержащихся в теле вызываемого модуля, сравнимо с числом команд, обеспечивающих связь модулей. Процедура сокращения числа вызовов подпрограмм является примером того, что два критерия оптимизации — время счета программы и объем занимаемой ею памяти — противоречат друг другу.

Помимо прямого оптимизационного эффекта следует также учитывать косвенный оптимизационный эффект, возникающий в результате выполнения рассматриваемой процедуры. Как пра-

вило, глобальная оптимизация программ практически для всех оптимизационных процедур более эффективна, если она выполняется над одним большим модулем, а не над эквивалентными ему (в функциональном смысле) несколькими маленькими модулями. Поэтому выполнение процедуры сокращения числа вызовов подпрограмм позволяет увеличить оптимизационный эффект других процедур, который может быть настолько существенным, что оправдывает выполнение рассматриваемой процедуры в большинстве случаев, даже если происходит дублирование тела вызываемого модуля.

В качестве примера, который поможет оценить величину косвенного оптимизационного эффекта, рассмотрим обобщение операций умножения «x», сложения «+» и равно «=» на матрицы. Операция умножения реализуется процедурой MULT, сложения — PLUS и равенства — ASSIGN (язык ПЛ/1):

```
MULT: PROCEDURE (X, Y, Z, L, M, N);
DECLARE (X (L, M), Y (M, N), Z (L, N)) FIXED (15. 3),
(L, M, N) FIXED (3. 0);
DO I = 1 TO L;
DO J = 1 TO N;
Z (I, J) = 0;
DO K = 1 TO M;
Z (I, J) = X (I, K) * Y (K, J) + Z (I, J);
END;
END;
END;
END MULT;

PLUS: PROCEDURE (X, Y, Z, L, M);
DECLARE (X (L, M), Y (L, M), Z (L, M)) FIXED (15. 3),
(L, M) FIXED (3. 0);
DO I = 1 TO L;
DO J = 1 TO M;
Z (I, J) = X (I, J) + Y (I, J);
END;
END;
END PLUS;
ASSIGN: PROCEDURE (X, Z, L, M);
DECLARE (X (L, M), Z (L, M), FIXED (15. 3),
(L, M) FIXED (3. 0);
DO I = 1 TO L;
DO J = 1 TO M;
Z (I, J) = X (I, J);
END;
END;
END ASSIGN;
```

Пусть требуется вычислить элементы матрицы D, равной A * B + C, где A, B, C — матрицы размером 20 × 20. В общем случае участок программы, вычисляющий матрицы D, имеет следующий вид:

```
а) CALL MULT (A, B, P1, 20, 20, 20)
   CALL PLUS (P1, C, P2, 20, 20)
   CALL ASSIGN (P2, D, 20, 20)
```

Здесь помимо затрат времени, обусловленных вызовом процедур, требуется дополнительно 800 слов памяти для временных массивов P1 и P2.

Участок (б) является результатом выполнения процедуры сокращения числа вызовов подпрограмм над участком (а) (на участке (б) и далее описания массивов и переменных опущены):

```
б) DO I = 1 TO 20;
   DO J= 1 TO 20;
   P 1 (I, J) = 0;
   DO K = 1 TO 20;
   P 1 (I, J) = A (I, K) * B (K, J) + P1 (I, J);
   END;
   END;
   END;
   DO I = 1 TO 20;
   DO J = 1 TO 20;
   P 2 (I, J) = P 1 (I, J) + C (I, J);
   END;
   END;
   DO I = 1 TO 20;
   DO J = 1 TO 20;
   D (I, J) = P 2 (I, J);
   END;
   END;
```

Над тремя внешними циклами участка (б) можно выполнить два раза процедуру объединения циклов. В результате ее выполнения участок (б) примет следующий вид:

```
в) DO I = 1 TO 20;
   DO J = 1 TO 20;
   P 1(I, J) = 0;
   DO K = 1 TO 20;
   P 1 (I, J) = A (I, K) * B (K, J) + P 1 (I, J);
   END;
   P 2 (I, J) = P 1 (I, J) + C (I, J);
   D (I, J) = P 2 (I, J);
   END;
   END;
```

Так как массивы P1 и P2 являются временными и не используются вне тела цикла, то над участком (в) можно выполнить процедуру экономии памяти при обработке массивов (см. гл. 2.9) и заменить обращение к элементам массивов P1 и P2 обращением к скалярным переменным P11 и P22 соответственно. В результате выполнения процедуры экономии памяти при обработке массивов участок (в) принимает вид

```
г) DO I = 1 TO 20;
   DO J = 1 TO 20;
   P 11 = 0;
   DO K = 1 TO 20;
   P 11 = A (I, K) * B (K, J) + P 11;
   END;
   P 22 = P 11 + C (I, J);
   D (I, J) = P 22;
   END;
   END;
```

После выполнения процедуры удаления излишних операторов присваивания участок (г) преобразуется в участок (д):

```
д) DO I = 1 TO 20;
   DO J = 1 TO 20;
   P 11 = 0;
```

```
DO K = 1 TO 20;
P 11 = A (I, K) * B (K, J) + P 11;
END;
D (I, J) = P 11 + C (I, J);
END;
END;
```

Участок (е) является результатом выполнения процедуры сдвига базового значения переменной (см. гл. 2.10) над участком (д):

```
е) DO I = 1 TO 20;
   DO J = 1 TO 20;
   P 11 = C (I, J);
   DO K = 1 TO 20;
   P 11 = A (I, K) * B (K, J) + P 11;
   END;
   D (I, J) = P 11;
   END;
   END;
```

Здесь не производится выполнение процедуры устранения излишних операторов присваивания над операторами P11 = A (I, K) * B (K, J) + P11 и D (I, J) = P11, так как обращение к элементу массива внутри цикла, как правило, менее эффективно, чем обращение к этому же элементу массива вне цикла, даже если появится дополнительный оператор присваивания.

Сравнив участки (б) и (е), получим, что косвенный оптимизационный эффект процедуры сокращения числа вызовов подпрограмм в ряде случаев может быть значительным и может превышать прямой оптимизационный эффект, получаемый в результате выполнения этой процедуры.

Следует заметить, что наиболее часто процедура сокращения числа вызовов подпрограмм применяется при вызове стандартных подпрограмм или неявно вызываемых подпрограмм, например преобразования типов данных и т. д. В отдельных системах оптимизации программ процедура сокращения числа вызовов подпрограмм, ориентированная на конкретные типы подпрограмм, рассматривается как отдельные процедуры, для обозначения которых обычно применяется термин «in line код». Выполнение таких процедур осуществляется либо автоматически, либо по желанию программиста, который указывает соответствующие параметры при обращении к системе оптимизации. Рассмотрим пять процедур, являющихся частными случаями процедуры сокращения числа вызовов подпрограмм, но в системах оптимизации программ рассматривающихся как отдельные процедуры.

In line код для вызовов операторов ввода-вывода. При выполнении операторов ввода-вывода, как правило, происходят обращение к сервисным библиотекам и вызов из них некоторых подпрограмм. Суть рассматриваемой процедуры заключается в замене вызовов этих подпрограмм самими этими подпрограммами. Эта процедура обычно выполняется автоматически.

In line код для объединения списков формата и операторов ввода-вывода. Как правило, при выполнении операторов ввода-вывода обработка каждого элемента списка сопровождается

вызовом соответствующих сервисных подпрограмм и появлением дополнительных команд в объектной программе, обеспечивающих преобразование данных из внутреннего представления во внешнее и наборот. Суть рассматриваемой процедуры заключается в сопоставлении каждого элемента списка оператора ввода-вывода с соответствующим ему элементом конструкции FORMAT (или эквивалентной ей конструкции) в процессе трансляции, а не выполнения программы. Это достигается помещением соответствующих подпрограмм, организующих также сопоставление, в текст объектной программы. Выполнение этой процедуры возможно в том случае, если вид соответствующих списков позволяет установить соответствие между их элементами, в частности, должны отсутствовать повторения, заданные арифметическими выражениями, аргументы которых неизвестны на этапе трансляции. Выполнение процедуры in line код для объединения списков формата и операторов ввода-вывода позволяет сократить число вызовов соответствующих подпрограмм.

In line код для обработки строковых данных. Суть этой процедуры состоит в замене обращений к подпрограммам обработки строковых данных самими этими подпрограммами. К таким подпрограммам относятся подпрограммы выполнения конкатенаций, присваивания, изменения длины и т. д.

In line код для встроенных функций. В современных системах оптимизации программ вызовы многих встроенных функций заменяются соответствующими подпрограммами. Суть рассматриваемой процедуры состоит в выполнении таких замен. Вызовы одних встроенных функций выполняются всегда, для других такие замены осуществляются только в простых случаях. Как правило, программисту предоставляется возможность управлять процессом выполнения этой процедуры.

In line код для преобразования типов данных. Суть этой процедуры состоит в замене вызовов подпрограмм, выполняющих преобразования типов данных самими этими подпрограммами. Она, как правило, выполняется автоматически и не для всех типов данных. Для наиболее сложных преобразований типов данных эта процедура обычно не выполняется.

Организация новых модулей. Эта процедура — антипод процедуры сокращения числа вызовов подпрограмм, целью ее выполнения является сокращение объема памяти, занимаемого программой. Суть процедуры организации новых модулей состоит в поиске в программе одинаковых (с точностью до некоторых операторов или параметров) участков программы, в организации новых модулей (подпрограмм, подпрограмм-функций, процедур), соответствующих выделенным участкам, и в замене этих участков операторами вызова соответствующих модулей. Например, участок (б) является результатом выполнения процедуры организации новых модулей над участком (а), организованная новая

подпрограмма XYAR показана на участке (в):

```
а) X = X 1 ** 2 + X 1 ** 3 + X 1 ** 4
   Y = Y 1/2. + Y 1/3. + Y 1/4.
   :
   A = B ** 2 + B ** 3 + D
   R = E/2. + E/3. + E/4.
б) CALL XYAR (X, X 1, X 1, 0, Y, Y1)
   :
   CALL XYAR (A, B, 0, D, R, E)
в) SUBROUTINE XYAR (T 1, T 2, T 3, T 4, T 5, T 6)
   T 1 = T 2 ** 2 + T 2 ** 3 + T 3 ** 4 + T 4
   T 5 = T6/2. + T6/3. + T6/4.
   RETURN
   END
```

Таким образом, новые модули могут организовываться и для несовпадающих участков программы. Такая организация связана с дополнительными затратами памяти, которые тем не менее оправдываются выигрышем в объеме памяти, получаемым в результате замены несовпадающих участков программы оператором вызова соответствующей подпрограммы. Результатом выполнения процедуры организации новых модулей является сокращение объема памяти, занимаемой программой, и некоторое увеличение времени ее счета.

Реализация этой процедуры наиболее сложная по сравнению с остальными процедурами, так как здесь необходимо учитывать не только большое число управляющих и информационных связей между операторами и переменными программы, но и рассматривать множество алгебраических задач (связанных с преобразованиями арифметических выражений), решение которых необходимо для выделения одинаковых операторов и участков программы.

Уменьшение числа параметров подпрограмм. При обращении к подпрограммам и подпрограммам-функциям значительная часть времени, необходимого для организации их информационных связей с основной программой, потребляется процессов вызова параметров. Процедура уменьшения числа параметров подпрограмм позволяет сократить это время путем организации списка параметров в единый массив. Рассмотрим, например, следующие два участка:

```
а) SUBROUTINE A
   :
   CALL BB (C 1, C 2, C 3)
   :
   RETURN
```

```
б) SUBROUTINE A
   DIMENSION Q (3)
   COMMON/Y/C 1, C 2, C 3
   EQUIVALENCE (Q (1), C 1)
   :
```

```
END
SUBROUTINE BB (X 1, X 2, X 3)
X 1 = X 2 ** 2 + X 3 ** 2
CALL R (X 1)
RETURN
END
```

```
CALL BB (Q)
.
.
.
RETURN
END
SUBROUTINE BB (XX)
DIMENSION XX (3)
COMMON/Y/X 1, X2, X3
EQUIVALENCE (XX (1), X 1)
X 1 = X2 ** 2 + X 3 ** 2
CALL R (X 1)
RETURN
END
```

Участок (б) является результатом выполнения процедуры уменьшения параметров подпрограмм над участком (а). С помощью рассматриваемой процедуры число параметров в вызываемом сегменте можно уменьшить до одного. Однако это не всегда возможно. Например, если параметры вызываемой подпрограммы находятся в различных блоках конструкции COMMON, то уменьшить число параметров можно только до двух, до трех и т. д. Поэтому перед выполнением этой процедуры должен быть проведен анализ конструкций COMMON и EQUIVALENCE в вызывающем и вызываемых сегментах. Существуют трансляторы с ФОРТРАНа, где формальные параметры нельзя использовать в конструкции EQUIVALENCE. Для таких трансляторов рассматривается процедура не применяется, а может быть применена процедура организации подпрограмм без параметров.

Организация подпрограмм без параметров. Процедура уменьшения числа параметров подпрограмм позволяет уменьшить число обращений к параметрам до минимума. Однако в результате ее выполнения нельзя в общем случае полностью устранить лишние вычисления и временные затраты, связанные с необходимостью установления информационных связей между вызывающим и вызываемым модулями. Результатом выполнения процедуры организации подпрограмм без параметров является установление информационных связей между основной программой и подпрограммами с помощью конструкции COMMON, что приводит к уменьшению времени выполнения подпрограмм. Формальные параметры подпрограммы описываются в отдельном блоке конструкции COMMON в вызываемом и вызывающем модулях. В результате этого во время выполнения подпрограммы при использовании переменной, являющейся до преобразования формальным параметром, происходит обращение к аппарату связи с основной программой параметров конструкции COMMON, который эффективней аппарата связи с основной программой формальных параметров. Участок (б) является результатом выполнения процеду-

ры организации подпрограмм без параметров над участком (а):

```
а) SUBROUTINE A (Y 1)
     :
   CALL B (C 1, C 2, C 3)
     :
   END
   SUBROUTINE B (C1, C2, C3)
     :
   END
```

```
б) SUBROUTINE A (Y 1)
   COMMON/XX/C 1, C 2, C 3
     :
   CALL B
     :
   END
   SUBROUTINE B
   COMMON/XX/C1 , C 2, C 3
     :
   END
```

Так как аппарат связи с основной программой используется при каждом обращении к любому формальному параметру подпрограммы, то выигрыш во времени, получаемый в результате выполнения рассматриваемой процедуры, зависит не от числа формальных параметров, а от общего числа обращений к этим параметрам.

Помимо прямого оптимизационного эффекта, получаемого в результате выполнения процедуры организации подпрограмм без параметров, существует еще косвенный оптимизационный эффект, связанный с отказом использования аппарата связи формальных параметров с основной программой. Выполнение рассматриваемой процедуры позволит разгрузить быстрые регистры, которые используются в этом аппарате, и, следовательно, даст возможность более эффективно использовать их для других целей, что, в свою очередь, ускорит выполнение программы. Все сказанное относительно подпрограмм в такой же мере относится к подпрограммам-функциям. Выполнение процедуры организации подпрограмм без параметров наряду с сокращением времени счета подпрограмм вызывает уменьшение объема памяти, занимаемого этими подпрограммами.

Замена параметров локальными переменными. Подпрограммы типа SUBROUTINE или FUNCTION, как правило, вызывают параметры по имени. Это обозначает, что при каждом обращении к параметру подпрограмма требует его внешний адрес, т. е. вызывается аппарат связи с основной программой. Для большинства трансляторов целесообразно преобразовать все скалярные параметры в локальные переменные подпрограммы и затем использовать локальную копию параметра при каждом обращении к нему, что сокращает число обращений к скалярным параметрам, т. е. число вызовов аппарата связи с основной программой. Процедура замены параметров локальными переменными позволяет свести число обращений к параметрам к минимуму. По характеру использования формальные параметры подпрограмм можно разделить на две группы. К первой группе относятся параметры,

которые являются входными данными для вызываемой подпрограммы, ко второй — которые являются выходными данными подпрограммы. Один и тот же параметр может принадлежать к первой и ко второй группам одновременно. Выполнение процедуры замены глобальных параметров локальными можно разбить на четыре этапа.

1. Каждому скалярному формальному параметру подпрограммы однозначно ставится в соответствие локальная переменная. Идентификаторы локальных переменных выбираются отличными от идентификаторов всех переменных, употребляемых в подпрограмме.

2. Идентификаторы скалярных формальных параметров в заголовке подпрограммы заменяются идентификаторами соответствующих им локальных переменных.

3. Каждому скалярному формальному параметру ставится в соответствие оператор присваивания. Для параметров первой группы его левой частью является идентификатор формального параметра, а правой частью — идентификатор соответствующей локальной переменной. Для второй группы, наоборот, левая часть — локальная переменная, правая часть — формальный параметр.

4. Операторы присваивания, соответствующие формальным параметрам первой группы, вставляются в тело подпрограммы перед первым выполняемым оператором. Если таких операторов несколько, то вставка производится перед каждым из них. Операторы присваивания, соответствующие формальным параметрам второй группы, вставляются в тело подпрограммы перед всеми операторами выхода из подпрограммы. Для формальных параметров, принадлежащих к двум граппам одновременно, соответствующие операторы присваивания вставляются как перед первыми выполняемыми операторами, так и перед операторами выхода из подпрограммы.

Например, участок (б) является результатом выполнения процедуры замены параметров локальными переменными над участком (а):

```
а) SUBROUTINE KK (A, B, C)   б) SUBROUTINE KK (X1, X2, X3)
   .                                A = X 1
   .                                B = X 2
   RETURN                           .
   END                              .
                                    X 2 = B
                                    X 3 = C
                                    RETURN
                                    END
```

Здесь предполагается, что подпрограмма KK имеет один оператор входа в подпрограмму и один оператор выхода из под-

программы; формальный параметр А относится к первой группе, С — ко второй и В — к двум одновременно.

Как правило, нецелесообразно преобразовывать скалярный параметр в локальную переменную, если к параметру обращаются один или два раза. Нецелесообразно преобразовывать параметры-массивы в локальные массивы, особенно для больших массивов, так как затраты ресурсов ЭВМ на такие преобразования превосходят их сокращение, обусловленное уменьшением обращений к аппарату связи с основной программой. Для некоторых типов ЭВМ имеет смысл выполнять аналогичную процедуру для глобальных переменных, описанных в конструкции COMMON.

Процедура замены параметров локальными переменными аналогично выполняется над программами-функциями.

Свертка формальных параметров. Эта процедура основывается на некоторых контекстных условиях обращения к подпрограммам, которые в ряде случаев позволяют уменьшить число формальных параметров подпрограмм. Если в основной программе имеется несколько обращений к подпрограмме, таких, что ряд фактических параметров в этих обращениях совпадает, рассматриваемая процедура в подобных случаях уменьшает число формальных параметров вызываемой подпрограммы за счет тех, которым соответствуют совпадающие во всех вызовах фактические параметры. При этом различают два случая. Совпадающие фактические параметры являются числовыми константами или переменными.

В первом случае формальные параметры, соответствующие совпадающим фактически параметрам, заменяются в теле подпрограммы соответствующими числовыми константами, и удаляются из заголовка и вызовов подпрограммы.

Во втором случае с помощью конструкций, описывающих глобальные переменные (типа COMMON в ФОРТРАНе), совпадающие фактические параметры объявляются такими переменными, затем устраняются из вызовов и заголовка вызываемой процедуры. Например, участок (б) является результатом выполнения процедуры свертки формальных параметров над участком (а):

```
a) SUBROUTINE A (Y1)
      .
      .
      .
   CALL BB (C, 2. 1, B, 1. 5)
      .
      .
      .
   CALL BB (R, E, B, 1. 5)
      .
      .
      .
   END
   SUBROUTINE BB (X 1, X 2, X 3, X4)
   X 1 = X 2 ** 2 + X 3 ** 2 + X 4 ** 2
   RETURN
   END
```

```
б) SUBROUTINE A (Y 1)
   COMMON B
   .
   .
   CALL BB (C, 2. 1)
   .
   .
   CALL BB (R, E)
   .
   .
   END
   SUBROUTINE BB (X 1, X 2)
   COMMON B
   X1 = X 2 ** 2 + B ** 2 + 2.25
   RETURN
   END
```

Оптимизационный эффект выполнения процедуры обусловлен сокращением числа обращений к фактическим параметрам во время выполнения вызываемой подпрограммы, т. е. сокращением числа обращений к подпрограммам связи вызываемого и вызывающего модулей. Кроме того, возникает косвенный оптимизационный эффект, обусловленный тем, что после выполнения процедуры свертки формальных параметров может появиться возможность выполнения других оптимизационных процедур, например выполнения константных действий, отождествления переменных и т. д. В частности, на участке (б) после выполнения процедуры свертки формальных параметров появилась возможность применения (и была применена) процедуры выполнения константных действий.

Оптимизация вычислений подпрограмм. Значения, присваиваемые переменным в результате выполнения подпрограмм, как правило, определяются только значениями фактических параметров этих подпрограмм. Поэтому, если в программе появляется вызов подпрограммы со списком фактических параметров, эквивалентным списку фактических параметров одного из предыдущих вызовов, то подпрограмма не обязательно должна быть вызвана, а может быть использован результат предыдущего вычисления. Суть процедуры оптимизации вычислений подпрограмм заключается в использовании результатов предыдущих вычислений подпрограмм. Например, в результате применения рассматриваемой процедуры к участку (а)

```
а) K 1 = 1
   .
   .
   CALL A (K1, 2, 3)
   .
   .
   K 3 = 3
   .
   .
   CALL A (1, 2, K3)
```

выполнение оператора CALL A (1, 2, K3) будет заключаться не в вычислении подпрограммы A, а в использовании соответствующих результатов, полученных при выполнении оператора CALL A (K 1, 2, 3) (здесь предполагается, что участки программ между соответствующими операторами присваивания и операторами вызова подпрограмм являются линейными).

Условия выполнения процедуры оптимизации вычислений подпрограмм, в общем случае, являются достаточно сложными. Это обусловлено тем, что для большинства языков программирования один и тот же формальный параметр может являться как входной, так и выходной переменной для вызываемой подпрограммы. Для выделения таких формальных параметров требуется глобальный анализ управляющих и информационных связей программы. Так, для приведенного участка (а), если параметры K1 и/или K3 являются входными и выходными параметрами подпрограммы A и их значения после выполнения подпрограммы A не равны 1 и/или 3 соответственно, то выполнение рассматриваемой процедуры может привести к изменению смысла участка (а). Кроме того, при выполнении процедуры оптимизации вычислений подпрограмм следует учитывать наличие глобальных переменных типа COMMON, которые в некотором смысле можно также рассматривать как формальные параметры подпрограмм. Особый случай представляют подпрограммы, которые используют функции типа датчиков случайных чисел. Значения выходных параметров таких подпрограмм даже при одинаковых значениях входных параметров, как правило, различны. Поэтому к таким программам рассматриваемая процедура не применяется.

Реализация процедуры оптимизации вычислений подпрограмм заключается в построении векторов входных и выходных параметров подпрограмм и установлении соответствия между значениями этих векторов. При выполнении рассматриваемой процедуры по значению вектора входных параметров находится значение вектора выходных параметров, которое и является результатом выполнения подпрограммы. Если соответствующего значения вектора входных параметров не существует, то происходит вычисление подпрограммы и запоминание значения вектора входных параметров и соответствующего ему значения вектора выходных параметров. Выполнение процедуры оптимизации вычислений подпрограмм целесообразно, в основном, если обращение к подпрограмме происходит чаще, чем изменение ее входных параметров.

Чистка рекурсивных областей. Эта процедура относится только к языкам высокого уровня, таким, как АЛГОЛ-60 и другим, в которых существует возможность рекурсивного вызова процедур. Рассматриваемая процедура аналогична процедуре чистки циклов. Суть ее состоит в выделении в теле рекурсивной процедуры рекурсивной области и операторов, рекурсивное выполнение

которых является функционально излишним. Выделенные операторы помещают до или после рекурсивной области. В результате выполнения процедуры чистки рекурсивных областей рекурсивная процедура преобразуется в процедуру, в общем случае состоящую из трех частей: операторов, рекурсивное выполнение которых является функционально излишним, и выполняемых до рекурсивной области; рекурсивной области; операторов, рекурсивное выполнение которых является функционально излишним, и выполняемых после рекурсивной области. В результате выполнения рассматриваемых преобразований над некоторой рекурсивной процедурой при каждом рекурсивном обращении к этой процедуре происходит вызов только выделенной рекурсивной области, а остальная часть процедуры выполняется только один раз при первом обращении к ней.

Выделение рекурсивных процедур. Аппарат реализации рекурсивных процедур содержит ряд специальных действий (в частности, копирование и защиту тела процедуры), обеспечивающих выполнение рекурсии и правильное выполнение этих процедур. Для нерекурсивных процедур такие действия являются функционально излишними, и их выполнение понижает качество программы. Процедура выделения рекурсивных процедур идентифицирует в программе рекурсивные и нерекурсивные процедуры и на основе этой идентификации определяет способ их реализации. В качестве одного из алгоритмов идентификации рекурсивных процедур может служить следующий.

Строится ориентированный граф G, вершинами которого являются имена используемых в программе процедур. От вершины *N1* идет стрелка к вершине *N2*, если в теле процедуры *N1* имеется прямое, или через фактический параметр вызова процедуры *N1*, обращение к процедуре *N2*. Процедуры, соответствующие вершинам графа *G*, через которые происходит хотя бы один контур, объявляются рекурсивными.

Этот алгоритм является приближенным, так как среди объявленных рекурсивными могут быть и нерекурсивные процедуры, однако все рекурсивные процедуры при использовании этого алгоритма будут объявлены рекурсивными. Для поиска вершин, через которые проходит хотя бы один контур, можно использовать алгоритм, приведенный в работе [47].

7. ИСПОЛЬЗОВАНИЕ ЭФФЕКТИВНЫХ ОПЕРАЦИЙ

Для сокращения времени счета программы и/или объема занимаемой ею памяти оптимизационные процедуры одних групп операций заменяют другими, которые в определенном контексте являются более эффективными.

Замена деления. Одной из самых медленно выполняемых команд ЭВМ является деление. Как правило, время выполнения этой команды превышает время выполнения команды умножения в несколько раз. Процедура замены деления в программе, написанной на языке высокого уровня, заменяет операцию деления операцией умножения. По характеру выполняемых преобразований эту процедуру можно разделить на две: локальную и глобальную.

Локальная процедура замены деления выполняется в пределах оператора только в том случае, когда делителем является

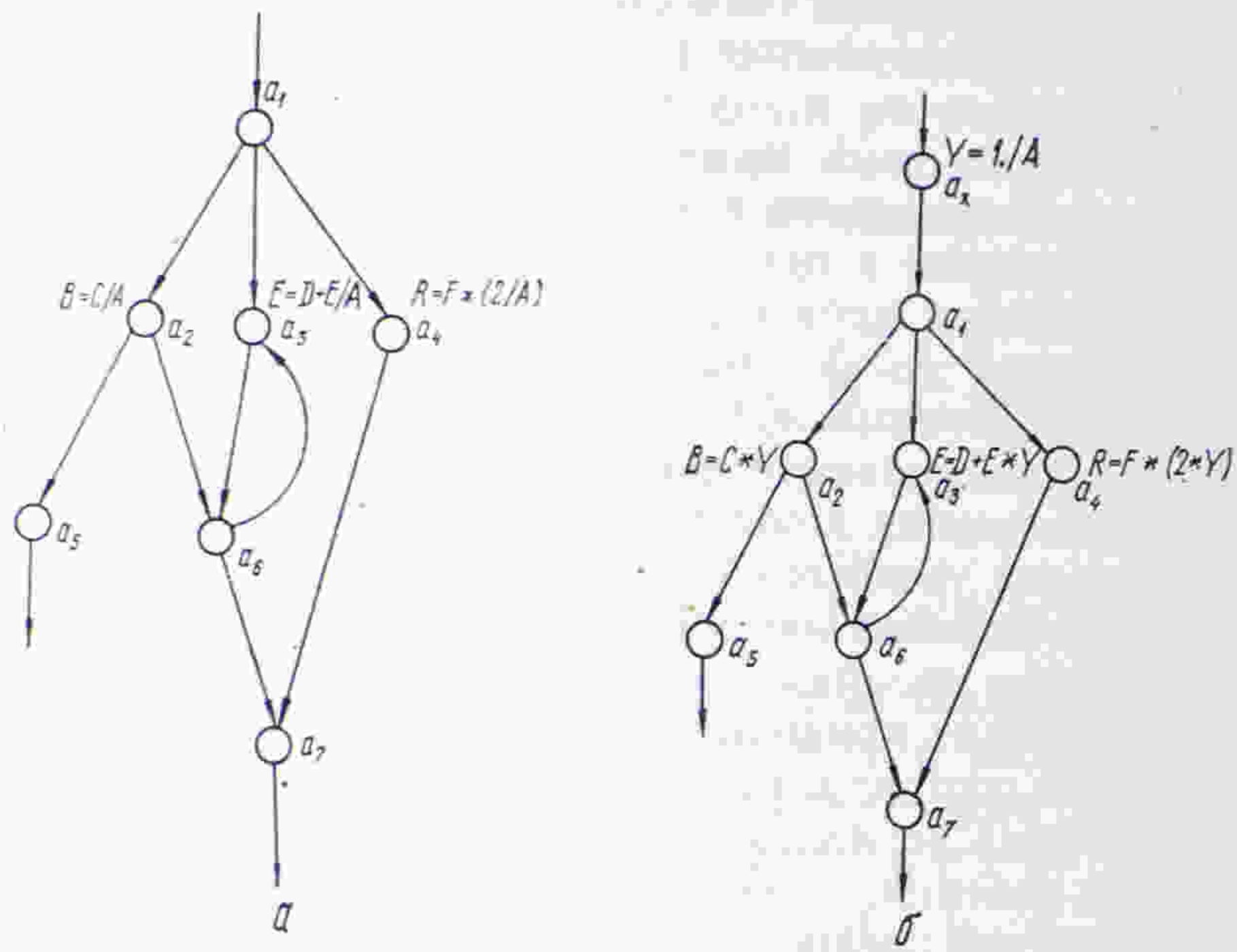


Рис. 19

число. Так, оператор A1 = A/4. в результате выполнения рассматриваемой процедуры преобразуется в оператор A1 = A * * 0.25.

Выполнение глобальной процедуры замены деления осуществляется в пределах всей программы и сводится в замене конструкций вида /X на конструкции вида * *Y*, где *Y* = 1./X; X — идентификатор переменной. При этом необходимо, чтобы в программе присутствовали хотя бы две конструкции вида/X (как правило, время выполнения двух команд деления превосходит время выполнения команды деления, засылки в ячейку и двух команд умножения) и выполнялись условия, определенные на основе анализа управляющего и информационного графов программы.

Граф, показанный на рис. 19, *б*, является результатом выполнения глобальной процедуры замены деления над графом, показанным на рис. 19, *а*. Здесь оптимизационный эффект возникает

вследствие того, что время выполнения оператора $Y = 1./A$ и трех операций умножения меньше времени выполнения трех операций деления. Процедура замены деления сокращает время выполнения программы, причем количественный эффект зависит от типа делителя. Наибольший выигрыш достигается тогда, когда делитель есть комплексная величина или числовое значение с двойной точностью, а сама операция находится в теле цикла.

Реализация умножения посредством сдвига. Выполнение процедуры реализации умножения посредством сдвига возможно только для ЭВМ, имеющих команду сдвига влево. Суть рассматриваемой процедуры состоит в реализации выражений вида X * * 2, X * 4, X * 8 и т. д., у которых один из сомножителей является степенью двух не посредством команд умножения, а посредством команд сдвига влево. Применение процедуры предполагает выполнение в процессе трансляции программы дополнительного анализа произведений нескольких сомножителей с целью выявления среди них сомножителей, кратных степени двух. Оптимизационный эффект, полученный в результате выполнения рассматриваемой процедуры, основывается на том, что время выполнения команд сдвига влево меньше времени выполнения команд умножения.

Реализация деления посредством сдвига. Выполнение процедуры реализации деления посредством сдвига возможно только для ЭВМ, имеющих команды сдвига вправо. Суть рассматриваемой процедуры состоит в реализации выражений вида X/2; X/4; X/8..., у которых делитель является степенью двух не посредством команд деления, а посредством команд сдвига вправо. Применение процедуры предполагает выполнение в процессе трансляции программы дополнительного анализа арифметических выражений с целью выявления делителей, кратных степени двух. Оптимизационный эффект, получаемый в процессе выполнения рассматриваемой процедуры, основывается на том, что время выполнения команд сдвига вправо меньше времени выполнения команд деления. Следует отметить, что процедура реализации деления посредством сдвига является альтернативной процедуре замены деления. Выбор одной из этих процедур определяется характеристиками конкретной ЭВМ.

Замена умножения. Практически во всех ЭВМ как универсальных, так и специализированных различные модификации команды умножения, как правило, выполняются медленнее аналогичных команд сложения. Процедура замены умножения, учитывающая это, заменяет операцию целочисленного умножения соответствующей ей операцией сложения. В простейшем случае после выполнения рассматриваемой процедуры выражение B * 2 заменяется выражением B + B. Например, участок (б) является результатом выполнения процедуры замены умножения над

участком (а):

```
а) A 1 = 3. 0 * B ** 2/C * K + SIN (X)
   A 2 = X1/B * E ** 2 * 2. + R * 3.
б) A 1 = (K + K + K) * B ** 2/C + SIN (X)
   A 2 = (X 1 + X 1)/B * E ** 2 + R + R + R.
```

Кроме того, если время выполнения команды умножения превосходит суммарное время одной или нескольких команд сложения и команд пересылки информации из ОЗУ в сумматор и обратно, то эффективное выполнение процедуры замены умножения возможно даже в случае, если один из сомножителей является нетривиальным арифметическим выражением, а другой—целым числом. Например, участок (г) является результатом выполнения процедуры замены умножения над участком (в) (причем во втором операторе участка (в) процедура замены умножения выполнена дважды):

```
в) A 1 = (B + C + D * K)/R * 2 + R
   A 2 = 3 * E ** 2 * (E/5. + B * 2) + SIN (X)
г) Y 1 = B + C + D * K
   Y 2 = E/5. + B + B
   A 1 = (Y 1 + Y 1)/R + R
   A 2 = E ** 2 * (Y2 + Y2 + Y2) + SIN (X)
```

Если целое число, являющееся одним из сомножителей, достаточно большое, то автоматически вместе с рассматриваемой процедурой выполняется процедура оптимизации выражений, например участок (д) преобразуется в участок (е):

```
д) A = B * 5          е) Y 1 = B + B
                         A = Y 1 + Y 1 + B
```

Иногда применяется другой вариант этой процедуры, когда операторы вида C = B * 3 заменяются последовательностью операторов

```
C = B + B
C = C + B
```

При увеличении значения числового сомножителя число машинных команд, реализующих соответствующие операции сложения, возрастает, т. е. возрастает суммарное время выполнения этих команд. Поэтому существует максимальное значение числового сомножителя, при котором выполнение процедуры замены умножения еще сохраняет смысл и приводит к положительному оптимизационному эффекту. Это максимальное значение зависит от вида сомножителя — арифметического выражения (в простейшем случае переменной), используемых методов трансляции, характеристик ЭВМ и определяется, как правило, эмпирическим путем.

Процедура замены умножения не выполняется для индексов. Кроме того, эта процедура не выполняется для выражений вида X * 2; X * 4; X * 8..., когда один из сомножителей является степенью двух, и используется процедура реализации умножения посредством сдвига.

При выполнении этой процедуры выигрыш по быстродействию, как правило, влечет за собой проигрыш по объему памяти, занимаемой программой.

Замена возведения в степень. Операция возведения в степень при трансляции в объектные коды, как правило, осуществляется путем вызова специальных подпрограмм. При этом практически не учитывается, в каком виде задан показатель степени, т. е. является ли он целым числом или дробным, что может существенно влиять на алгоритм, а следовательно, и время выполнения этой операции. Суть процедуры замены возведения в степень состоит в замене операции возведения в степень (когда показатель степени является целым числом) операцией (операциями) умножения. Участок (б) является результатом выполнения рассматриваемой процедуры над участком (а):

```
а) Y 1 = A + B ** 3          б) Y 1 = A + B * B * B
   Y 2 = C/D ** E ** 2          Y 2 = C/D ** (E * E)
```

Если время выполнения команды возведения в степень превосходит суммарное время выполнения одной или нескольких команд умножения и команд пересылки информации из ОЗУ в сумматор и обратно, то эффективное выполнение процедуры замены возведения в степень возможно в случае, если основанием степени является нетривиальное арифметическое выражение любой сложности, а показатель степени является целым числом. Например, участок (г) является результатом выполнения процедуры замены возведения в степень над участком (в) (причем во втором операторе участка (в) процедура замены возведения в степень выполнена несколько раз):

```
в) B 1 = (E + D/5. — C) ** 2 + X
   B 2 = X 1/((X 2 ** 3 + 2.) ** 3 + (X 3 + X 4) ** 2)

г) Y 1 = E + D/5. — C
   Y 2 = X 2 * X 2 * X 2 + 2.
   Y 3 = X 3 + X 4
   B 1 = Y 1 * Y 1 + X
   B 2 = X 1/(Y 2 * Y 2 * Y 2 + Y 3 * Y3)
```

Если показатель степени является достаточно большим целым числом, то автоматически вместе с рассматриваемой процедурой выполняется процедура оптимизации выражений. Например, оператор участка (д) преобразуется в операторы участка (е):

```
д) A = B ** 5          е) Y 1 = B * B
                          A = Y 1 * Y 1 * B
```

Иногда применяется другой вариант этой процедуры, когда операторы вида D = E ** 3 заменяются последовательностью операторов

```
D = E * E
D = D * E
```

При увеличении значения показателя степени число машинных команд, реализующих соответствующие операции умножения, возрастает, т. е. возрастает суммарное время выполнения этих команд. Поэтому существует максимальное значение показателя степени, при котором выполнение процедуры замены возведения в степень еще сохраняет смысл. Если значение показателя степени превышает максимальное, то выполнение расматриваемой процедуры теряет смысл и приводит к отрицательному оптимизационному эффекту. Максимальное значение показателя степени, при котором еще достигается положительный эффект в результате выполнения процедуры замены возведения в степень для каждой конкретной ЭВМ и конкретного транслятора так же, как и в процедуре замены умножения, определяется путем машинного эксперимента. Выигрыш во времени в результате выполнения замены возведения в степень возникает вследствие разницы времен выполнения подпрограмм, осуществляющих возведение в степень, и выполнения нескольких команд умножения и команд пересылки информации из ОЗУ на сумматор и обратно, осуществляющих операцию перемножения соответствующих сомножителей.

Замена последовательных операций деления на умножение. В программах встречаются подвыражения, содержащие несколько последовательных операций деления, где каждое делимое, выполняющее в этом случае одновременно и функцию делителя, является переменной, функцией или подвыражением, заключенным в скобки. В таких случаях существует возможность замены одной или нескольких последовательных операций деления операциями умножения, что и осуществляет рассматриваемая процедура. Например, участок (б) является результатом выполнения процедуры замены последовательных операций деления на умножение над участком (а):

```
а) A 1 = B/D/E + C                б) A1 = B/(D * E) + C
   :                                 :
   A 2 = 3. * B/(A + B)/C + E        A 2 = 3.* B/((A + B) * C)+ E
```

Для повышения оптимизационного эффекта этой процедуры предварительно необходимо выполнить переупорядочение операндов арифметических выражений таким образом, чтобы операции деления по возможности раполагались последовательно, например подвыражения типа A/B * D ** 2/3 преобразовать в подвыражения типа A/B/3 * D ** 2. Однако при этом

следует учитывать специфику вычисления на ЭВМ больших и малых чисел и чисел типа INTEGER. Оптимизационный эффект, возникающий в результате выполнения рассматриваемой процедуры, обусловлен разницей между временами выполнения одной или нескольких команд деления и такого же числа команд умножения и команды пересылки информации из ОЗУ в сумматор, как правило, появляющейся вследствие использования скобок.

Упрощение выражений. Рассматриваемая процедура упрощает арифметические выражения языков высокого уровня, преобразуя их к виду, который позволяет генерировать более эффективный объектный код. В частности, если это возможно, операция умножения заменяется операцией сложения, которая, как правило, выполняется на ЭВМ быстрее. Участок (б) является результатом выполнения процедуры упрощения выражений над участком (а) (язык ПЛ/1):

```
а) DO I = 2 TO N BY 2;
   :
   A (I) = I * 4;
   :
   END;
б) I = 2;                    (Инициализация цикла)
   IF I > N THEN GOTO F;
   Y 1 = 4 * I;
G: :                         (Вход в цикл)
   A (I) = Y 1;
   :
   I = I + 2;                (Проверка условий
   Y 1 = Y 1 + 8             выполнения цикла)
   IF I < N THEN GOTO G
F: :                         (Выход из цикла)
```

Если к участку (б) применить последовательно процедуры подстановки и выполнения континентных действий, то оператор Y1 = 4 * I заменяется оператором Y1 = 8, что приводит к дополнительной экономии времени счета и объема памяти, занимаемой рассматриваемым участком.

Процедура упрощения выражений позволяет учесть более тонкие особенности программ, а следовательно, получить более значительный оптимизационный эффект, чем аналогичная ей процедура замены умножения. Процедура замены умножения преобразует участок (а) в участок (в):

```
в) DO I = 2 TO N BY 2;
   :
   A (I) = I + I + I + I;
   :
   END;
```

На участке (в) для вычисления значения переменной А (I) требуется три операции сложения, в то время как на участке (б) — только одна операция сложения. Запись участка (б) сложнее записи участков (а) и (в), что обусловлено явным описанием действий, выполняемых циклом.

Оптимизация через «щель». Эта процедура выполняет локальную оптимизацию программ, записанных на языках различных уровней (как программ, записанных на языках высокого уровня, так и объектных программ, полученных в результате генерации). Суть ее заключается в просмотре нескольких рядом стоящих операторов (команд) как бы через скользящую по тексту программы «щель» с целью выявления, а затем замены отдельных групп операторов (команд) другими, более эффективными, в частном случае, просто удаления отдельных операторов (команд). Такие группы операторов (команд) выявляют при сравнении их с шаблонами, для каждого из которых указана необходимая замена. Такие шаблоны, называемые основными, определяются, как правило, конкретными свойствами языка, на уровне которого выполняется рассматриваемая процедура.

Ниже приведено несколько наиболее общих шаблонов. Они записаны на ассемблере, где ЗАП А — оператор пересылки содержимого сумматора в ячейку А; ЗАГР А — оператор пересылки содержимого ячейки А в сумматор; СЛОЖ А — оператор сложения содержимого сумматора с содержимым ячейки А; УМНЖ А — оператор перемножения содержимого сумматора и содержимого ячейки А; ВЫЧТ А — оператор вычитания из содержимого сумматора содержимого ячейки А.

1. Последовательность операторов вида

ЗАП А
ЗАГР А

удаляется из программы при условии, что либо ячейка А не будет использоваться далее, либо перед использованием ячейка А будет заполнена заново, и на эти операторы нет передач управления. Чаще всего можно удалить лишь один оператор ЗАГР А. Для этого только требуется, чтобы на него не было передач управления из других частей программы.

2. Последовательность операторов вида

ЗАГР А
ЗАП В

удаляется из программы, если за ней следует другой оператор ЗАГР и нет переходов на операторы ЗАП А. Кроме того, все последующие использования В будут заменены А до тех пор, пока не появится другой оператор ЗАП В (это преобразование выполняется для линейного участка). Рассматриваемый шаблон в некотором смысле аналогичен оптимизационной процедуре отождествления переменных.

3. Для языков высокого уровня медленно выполняемая операция вида А ** 0.5 заменяется быстро выполняемой операцией SQRT (А).

Кроме того, имеются вспомогательные шаблоны, с помощью которых осуществляется предварительное преобразование программ без повышения их качества к виду, удобному для применения основных шаблонов и выполнения других оптимизационных

преобразований. Применение вспомогательных шаблонов предполагает последующее применение основных шаблонов, с помощью которых осуществляется собственно оптимизация, т. е. замена одних групп команд другими, более эффективными.

Рассмотрим два вспомогательных шаблона, которые, как правило, применяются вместе с первым и вторым рассмотренными основными шаблонами.

1. Последовательность операторов вида

```
ЗАГР  A
СЛОЖ  B
```

заменяется последовательностью операторов

```
ЗАГР  B
СЛОЖ  A
```

При этом предполагается, что операция СЛОЖ коммутативная и не существует передачи управления на оператор СЛОЖ B.

2. Аналогичным образом, если операция УМНЖ является коммутативной и не существует передачи управления на оператор УМНЖ B, последовательность операторов

```
ЗАГР  A
УМНЖ  B
```

заменяется последовательностью операторов

```
ЗАГР  B
УМНЖ  A.
```

На практике для конкретного языка множество шаблонов может оказаться достаточно обширным и применяются они в самых разных комбинациях.

Участок (б) является результатом выполнения процедуры оптимизации через «щель» над участком (а):

```
а) ЗАГР 3          б) ЗАГР  E
   ЗАП  X1            СЛОЖ  B
   ЗАГР E             УМНЖ  3
   ЗАП  X2            ЗАП  A
   ЗАГР B
   СЛОЖ  X2
   УМНЖ  X1
   ЗАП  A
```

При выполнении рассматриваемой процедуры над участком (а) были применены следующие шаблоны:

первый вспомогательный к последовательности операторов

```
ЗАГР  B
СЛОЖ  X2
```

первый основной к последовательности операторов

```
ЗАП  X2
ЗАГР  X2
```

второй основной к последовательности операторов

```
ЗАГР  3
ЗАП  X1
```

Для рассматриваемого участка возможны и другие последовательности применяемых шаблонов. Участок (а) является результатом трансляции фрагмента реальной программы

```
X1 = 3
X2 = E
A = (B + X2) * X1
```

записанной на языке высокого уровня, в язык ассемблера, и применение, на первый взгляд, простой процедуры оптимизации через «щель», позволило сократить число команд этого участка в два раза.

Наибольший эффект от выполнения процедуры оптимизации через «щель», как правило, получают за счет сокращения числа команд пересылки информации из ОЗУ и быстрых регистров на сумматор и обратно, что, в частности, достигается путем использования свойства коммутативности операций.

Использование логического оператора IF. Для большинства трансляторов и ЭВМ использование арифметического оператора IF оправдано только в тех случаях, когда необходим переход в три различные точки программы по результату сравнения некоторого арифметического выражения на «меньше», «равно», «больше». Во всех остальных случаях является более оправданным использование логического оператора IF. Процедура использования логического оператора IF заменяет арифметический оператор IF (там, где его применение не оправдано) логическим оператором IF. Например, в результате выполнения рассматриваемой процедуры участок (а) преобразуется в участок (б):

```
а) IF (E + 5.) 50, 40, 40        б) IF (E. LT. 5.) A = B ** 3
50A      = B ** 3                     C = D + F ** 2
40C      = D + F ** 2
```

Оптимизационный эффект, возникающий в результате выполнения процедуры использования логического оператора IF, обусловлен уменьшением в объектной программе числа команд передач управления и в исходной программе числа операторов с метками, что, вследствие размещения команд, соответствующих операторам с метками, в левой части ячейки, приводит для ряда ЭВМ к дополнительной экономии памяти.

Арифметический оператор IF не эффективен и с других точек зрения, в частности, он затрудняет читаемость и понимаемость программы, и его присутствие в языках программирования объясняется скорее всего историческими причинами [92].

Засылка единицы в регистр. Для некоторых ЭВМ с несколькими быстрыми регистрами выгодно поместить в один из этих регистров единицу. Это позволяет ускорить ссылки на константу 1 (единицу). По тем же причинам при частом использовании других констант выгодно помещать эти константы в быстрые регистры. Целесообразность выполнения рассматриваемой процедуры существенно зависит от характеристик используемой ЭВМ и конкретной выполняемой программы.

Использование языка ассемблера. Даже применение широкого набора оптимизационных процедур не позволяет в отдельных случаях получить необходимое качество объектных кодов, являющихся результатом трансляции программ, написанных на языках высокого уровня. Это обусловлено наличием «узких» мест в таких программах, т. е. участков, значительно ухудшающих качество программ. Наличие таких «узких» мест объясняется рядом причин, в частности, разнообразием неоптимальных конструкций, встречаемых в программах, некоторыми особенностями математического обеспечения и разнообразием внутренней структуры ЭВМ. Для того чтобы сохранить преимущества языков высокого уровня и получить необходимое качество трансляции, «узкие» места программ записывают на языках более низкого уровня, в частности, на языке ассемблера. Рассматриваемая процедура обеспечивает возможность включать в программы, записанные на языках высокого уровня, отдельные участки, записанные на языке ассемблера. Одним из наиболее эффективных случаев применения процедуры использование языка ассемблера является организация процесса ввода-вывода.

Использовать язык ассемблера при написании программ на языках высокого уровня надо крайне осмотрительно [91], так как частое применение этого приема может свести на нет все преимущества и эффективность языков высокого уровня.

8. ОПТИМИЗАЦИЯ ВВОДА-ВЫВОДА

Для вычислительных задач эффективная организация ввода-вывода имеет большое значение в связи с тем, что время выполнения одного оператора ввода-вывода превышает, как правило, на порядок время выполнения одного арифметического оператора. Поэтому правильная организация ввода-вывода может значительно сократить время выполнения программы. В системах реального времени [78] ввод-вывод не должен, в общем случае, рассматриваться как часть вычислительного процесса.

Объединение операторов ввода-вывода. При выполнении каждого оператора ввода-вывода происходит обращение к сервисной библиотеке системы, вызов некоторых подпрограмм, которые, в свою очередь, как правило, осуществляют вызов супервизора. Для сокращения числа таких обращений выполняется процедура объединения операторов ввода-вывода, которая уменьшает число и частоту выполнения этих операторов. Например, два последовательных оператора WRITE или PRINT объединяются в один. Участок (б) является результатом выполнения процедуры объединения операторов ввода-вывода над участком (а):

```
а) PRINT 3, A, B, C          б) PRINT 3, A, B, C, D, E, F
   PRINT 3, D, E, F
```

Конструкции FORMAT при выполнении рассматриваемой процедуры меняются соответствующим образом.

Результатом выполнения процедуры объединения операторов ввода-вывода является сокращение времени выполнения программы и объема занимаемой ею памяти. Последовательное выполнение процедуры объединения операторов ввода-вывода и процедуры in line код для вызовов операторов ввода-вывода позволяет свести затраты ресурсов ЭВМ, необходимые для выполнения операторов ввода-вывода, к минимуму.

Уменьшение числа единиц записей операторов ввода-вывода. При выполнении операторов ввода-вывода пересылка каждой единицы записи сопровождается вызовом соответствующих сервисных подпрограмм и появлением дополнительных команд в объектной программе, обеспечивающих преобразование данных из внутреннего представления во внешнее и наоборот. Под единицей записи здесь понимается элемент списка операторов ввода-вывода, который может быть скалярной переменной или массивом.

Процедура уменьшения числа единиц записей операторов ввода-вывода объединением (с помощью специальных конструкций, как правило, имеющихся в языках программирования) всех или нескольких единиц записей в единый массив сокращает число единиц записей, и, следовательно, число обращений к сервисным подпрограммам. Например, участок (б) является результатом выполнения рассматриваемой процедуры над участком (а):

```
а) DIMENSION A (30), B (55)
      :
   PRINT 3 R, A, B, C, K
б) DIMENSION A (30), B (55), F (88)
   COMMON/XX/R, A, B, C, K
   EQUIVALENCE (R, F (1))
      :
   PRINT 3, E
```

Заметим, что не всегда можно сократить число единиц записей до единицы, так как элементы списка операторов ввода-вывода могут быть описаны в различных блоках конструкции COMMON.

Результатом выполнения рассматриваемой процедуры является сокращение времени счета программы и уменьшение объема занимаемой ею памяти. Последовательное выполнение процедуры уменьшения числа единиц записей операторов ввода-вывода и процедуры in line код для объединения списков формата и операторов ввода-вывода позволит свести затраты ресурсов ЭВМ, необходимые для организации процесса преобразования типов данных из внешнего представления во внутреннее и наоборот, к минимуму.

Организация бесформатного ввода-вывода. Бóльшая часть времени организации ввода-вывода уходит на преобразование

информации из внутреннего представления во внешнее и наоборот. Характер таких преобразований определяется специальными предложениями, например FORMAT в ФОРТРАНе. Как правило, при использовании конструкций типа FORMAT информация преобразуется путем вызова сервисных подпрограмм. С помощью процедуры организации бесформатного ввода-вывода осуществляется отказ от использования предложений типа FORMAT, т. е. ввод-вывод информации выполняется в том виде, в каком она находится в ЭВМ, без преобразования во внешнее представление. Наиболее целесообразна организация в программах бесформатного ввода-вывода для временных файлов и для файлов, используемых только в качестве вводной информации для других программ. В результате выполнения рассматриваемой процедуры время обмена сокращается в несколько раз, в отдельных случаях в десятки раз.

9. ЭФФЕКТИВНОЕ ИСПОЛЬЗОВАНИЕ ПАМЯТИ

Рассмотрим оптимизационные процедуры, основной функцией которых является размещение переменных, констант и команд таким образом, чтобы они занимали минимальный объем памяти ЭВМ.

Глобальная экономия памяти. Суть этой процедуры заключается в уменьшении объема памяти путем совмещения некоторых переменных программы. Две переменные можно совместить, т. е. обозначить их одним идентификатором, если не совпадают их пути на графе программы от определения до использования значений этих переменных. Для построения таких путей требуется анализ всех управляющих и информационных связей программы.

Разработанные к настоящему времени алгоритмы реализации этой процедуры сводят ее к задаче раскраски графа [40, 44, 65]. Общее решение задачи раскраски графа [49] содержит большое число переборов, очень громоздко и вследствие этого трудно реализуемо на ЭВМ. Поэтому пользуются обычно упрощенными алгоритмами [75], которые дают приближенный результат, как правило, удовлетворяющий нужды практического программирования.

В связи с наличием различных уровней оптимизации программ следует заметить, что программа, записанная на входном процедурно-ориентированном языке, чаще всего содержит меньше переменных, чем соответствующий получаемый после трансляции объектный модуль. Это связано с тем, что объектные программы привязаны к конкретной ЭВМ, имеющей ограниченные ресурсы (число быстрых регистров, магазинов и т. д.). Ограниченность ресурсов приводит к появлению в объектной программе переменных: дополнительных, с помощью которых происходит распределение ресурсов ЭВМ, и рабочих, появляющихся в процессе трансляции. Поэтому целесообразно выполнение процедуры глобаль-

ной экономии памяти в несколько этапов (на входном, промежуточном и машинном языках). На первом этапе осуществляется оптимизация памяти, занимаемой переменными, имеющимися в программах, составленных на входном языке. На следующих этапах эта процедура выполняется над программами, записанными на промежуточных либо машинном языках, которые в результате преобразований, выполняемых на первом этапе, содержат меньше переменных, чем в случае одноэтапного выполнения этой процедуры.

Многоэтапное выполнение процедуры глобальной экономии памяти уменьшает число вершин в анализируемых графах, что позволяет использовать алгоритмы, дающие более точные результаты при решении задачи раскраски графа, т. е. повышает оптимизационный эффект этой процедуры.

Если при выполнении оптимизационных процедур требуется сохранить наглядность программы, т. е. по возможности произвести минимальное число изменений ее текста, то в таких случаях не обязательно производить переобозначение переменных. Этого можно избежать, используя специальные средства, имеющиеся в языках высокого уровня, позволяющие описывать переменные, помещаемые в одну ячейку (что эквивалентно обозначению переменных одним идентификатором). В качестве примера можно привести конструкцию EQUIVALENCE в ФОРТРАНе. Процедура глобальной экономии распределяет память, а с помощью специальных средств описывается это распределение.

Процедуру глобальной экономии памяти также осуществляют при размещении в ней микропрограмм [73] и при организации пересылок массивов из внешней памяти в ОЗУ и обратно [101].

Локальная экономия памяти. Под локальной экономией памяти понимается экономия памяти в пределах линейного участка. Рассматриваемая процедура в основном предназначена для экономии памяти, отводимой под промежуточные результаты (переменные), т. е. такие результаты, которые используются на данном линейном участке и не используются ни в одном другом линейном участке. Результатом выполнения этой процедуры является определение минимального числа идентификаторов, необходимых для обозначения всех промежуточных переменных. Так как идентификатор, как правило, обозначает определенную ячейку, то результатом выполнения рассматриваемой процедуры является определение минимального числа ячеек, необходимых для хранения промежуточных результатов.

Рассмотрим один из алгоритмов, реализующих локальную экономию памяти, согласно которому экономия осуществляется за один просмотр команд линейного участка, последовательно, начиная с последнего и кончая первым [97]. Результатом этого просмотра является заполнение некоторой таблицы T, в каждой строке которой записывается идентификатор промежуточной пере-

менной. Пусть рассматривается очередной оператор присваивания (линейный участок может содержать только операторы присваивания либо операторы, аналогичные им). Если в левой части этого оператора находится идентификатор промежуточной переменной, то в таблице T имеется строка, содержащая его, и очищается (здесь предполагается, что на рассматриваемом линейном участке все промежуточные переменные являются используемыми). Если в правой части рассматриваемого оператора присваивания используется промежуточная переменная, идентификатор которой не занесен ни в одну из строк таблиц T, то этот идентификатор заносится в самую верхнюю свободную строку таблицы T. Если в таблице T нет свободных строк, то к ней присоединяется еще одна свободная строка, в которую заносится рассматриваемый идентификатор. После заполнения таблицы T число ее строк равно минимальному числу идентификаторов, необходимых для обозначения промежуточных переменных, и все идентификаторы, которые были записаны в одной строке, могут быть заменены одним общим идентификатором. Промежуточные переменные для данного линейного участка находятся просто во время синтаксического анализа программы.

Рассмотренный алгоритм выполняется для всех линейных участков программ и затем находится минимальное количество идентификаторов, необходимых для обозначения всех промежуточных переменных программы. Если при выполнении процедуры локальной экономии памяти требуется сохранить наглядность программы, то, как и в случае процедуры глобальной экономии памяти, не обязательно производить переобозначение переменных, а можно пользоваться имеющимися в большинстве языков программирования средствами, позволяющими описывать переменные, помещаемые в одну ячейку.

Устранение общих констант. Если в программе некоторая константа используется более чем один раз, то рассматриваемая процедура сокращает объем памяти, занимаемой программой, за счет размещения в памяти только одной копии этой константы. Например, пусть в памяти будет размещена только одна копия константы 1:

$$A = B + C + 1$$
$$D = E - 1$$

Процедура устранения общих констант выполняется в основном на уровне машинных кодов.

Сокращение числа блоков COMMON. Рассматриваемая процедура экономит память для набора взаимосвязанных подпрограмм, преобразуя в каждой из подпрограмм этого набора часть локальных массивов и переменных в глобальные, т. е. они передаются в обще пользование. При этом часть или все блоки конструкций COMMON объединяются в один, для того чтобы этот блок

могли использовать подпрограммы, имеющие большие массивы. Оптимизационный эффект, возникающий в результате выполнения рассматриваемой процедуры, обусловлен, во-первых, размещением локальных переменных различных подпрограмм в одном участке памяти и, во-вторых, размещением части локальных переменных в памяти, отводимой под глобальные переменные. Выполняя процедуру сокращения числа блоков COMMON, следует учитывать, что преобразование переменных из локальных в глобальные приводит к увеличению времени обращения к этим переменным.

Устранение лишних меток. По ряду причин в программах пользователей встречаются метки перед операторами, на которые нет передач управления. Такие метки являются лишними, и их следует удалить из программы с помощью процедуры устранения лишних меток. Оптимизационный эффект, возникающий в результате выполнения этой процедуры, обусловлен спецификой выполнения некоторых команд на ЭВМ. Например, имеются ЭВМ, у которых в одной ячейке располагается две команды, и для таких ЭВМ команда, на которую передается управление, должна располагаться первой в ячейке. Следовательно, если в процессе трансляции не выполняется специальный синтаксический анализ, то команды, в которые транслируется оператор с меткой, размещаются с начала очередной ячейки. При этом вторая половина предыдущей ячейки (если она не занята) не используется.

Экономия памяти при обработке массивов. Суть этой процедуры заключается в следующем. Пусть в некотором цикле последовательно вычисляются элементы промежуточного массива X, затем в этом же цикле элементы массива *X* обрабатываются некоторым образом и в дальнейшем нигде не используются. Тогда при выполнении ряда условий, сохраняющих смысл цикла, можно заменить вычисления и обработку элементов промежуточного массива *X* вычислением и обработкой промежуточной скалярной переменной. Основные условия, определяющие возможность выполнения рассматриваемой процедуры, в общем виде можно сформулировать следующим образом:

каждый элемент промежуточного массива используется и/или определяется только на одной итерации цикла;

на каждой итерации цикла используется и/или определяется только один элемент промежуточного массива; другими словами, между итерациями цикла или, иначе говоря, между значениями управляющей переменной цикла и элементами промежуточного массива в некотором смысле можно установить взаимно-однозначное соответствие.

Например, участок (*б*) является результатом выполнения процедуры экономии памяти при обработке массивов над участком (а), где вычисление и обработка промежуточного массива C (I) заменяется вычислением и обработкой скалярной переменной **S**:

```
а) DO 1 K = 1, 50
   C(K) = B(K) + SIN(K * 0.01)
   :
   C(K) = C(K) ** 3 + 5
   :
1  E(K) = C(K) * 3. + A(K)
```

```
б) DO 1 K = 1, 50
   S = B(K) + SIN(K * 0.01)
   :
   S = S ** 3 + 5.
   :
1  E(K) = S * 3. + A(K)
```

Выполнение процедуры экономии памяти при обработке массивов сокращает объем памяти, занимаемой программой (за счет уменьшения числа ячеек, отводимых под промежуточные результаты), и время ее выполнения (за счет сокращения числа обращений к индексируемым переменным).

Если в ЭВМ принята страничная организация памяти, выполнение процедуры экономии памяти при обработке массивов приводит к дополнительному оптимизационному эффекту. Это связано с тем, что разные массивы, в частности промежуточные (на приведенном примере массив С (К)), могут располагаться на разных листах памяти. Обращение к таким массивам связано с переходом с одного листа памяти на другой Выполнение рассматриваемой процедуры сводит число таких переходов к минимуму, что приводит к дополнительному сокращению времени.

Ситуации, когда выполняются все условия, позволяющие применять процедуру экономии памяти при обработке массивов, наиболее часто возникают при обработке матриц в результате выполнения процедуры объединения циклов.

10. ВСПОМОГАТЕЛЬНЫЕ ПРОЦЕДУРЫ

Выполнение вспомогательных процедур позволяет получить только косвенный оптимизационный эффект, и поэтому предполагает последующее выполнение основных оптимизационных процедур (под основными здесь подразумеваются все процедуры, не являющиеся вспомогательными).

Сдвиг базового значения переменной. При программировании итерационных процессов, как правило, на первой итерации некоторой переменной определяется начальное значение, называемое базовым, и все значения этой переменной, получаемые на последующих итерациях, вычисляются относительно базового значения прибавлением к нему некоторой величины (или умножением его на некоторую величину), задаваемой в виде константы или арифметического выражения. Если после окончания итерационного процесса к значению рассматриваемой переменной прибавляется некоторая величина (или значение рассматриваемой переменной умножается на некоторую величину), то это можно рассматривать как сдвиг базового значения. Суть процедуры сдвига

базового значения переменной заключается в выделении в программе операторов, изменяющих на первой итерации и после выполнения итерационного процесса базовое значение переменной, и в замене их одним оператором, определяющим на первой итерации сдвинутое базовое значение этой переменной. Например, участок (б) является результатом выполнения рассматриваемой процедуры над участком (а):

```
а)  A = A0                    б)  A = A0 + R
    C = C0                        C = C0 * F
    DO 1 K = 1, 100               DO 1 K = 1, 100
    A = B(K) * K + A              A = B(K) * K + A
  1 C = C * D(K) ** 3           1 C = C * D(K) ** 3
    A = A + R                     A = A
    C = C * F                     C = C
```

Рассматриваемая процедура часто применяется вместе с процедурами устранения неиспользованных определений, выполнения константных действий, оптимизации выражений, удаления тождественных операторов, удаления излишних операторов присваивания. Например, после выполнения процедуры сдвига базового значения переменной на участке (б) появились операторы A = A и C = C, для удаления которых должна выполниться процедура удаления тождественных операторов. Кроме того, если на первой итерации переменной присваивается базовое значение 0 (или 1) и для вычислений всех последующих значений этой переменной к базовому значению прибавляется некоторая величина (или умножается базовое значение на некоторую величину), то в результате выполнения рассматриваемой процедуры можно получить прямой оптимизационный эффект. Участок (г) является результатом выполнения процедуры сдвига базового значения над участком (в):

```
в)  A = 0                     г)  A = R
    C = 1                         C = F
    DO 1 K = 1, 100               DO 1 K = 1, 100
    A = B(K) * K + A              A = B(K) * K + A
  1 C = C * D(K) ** 3           1 C = C * D(K) ** 3
    A1 = A + R                    A1 = A
    C1 = C * F                    C1 = C
```

Здесь в результате применения рассматриваемой процедуры получен прямой оптимизационный эффект за счет сокращения числа выполнений операций сложения и умножения.

Наиболее часто процедура сдвига базового значения переменной применяется к рекурсивно-определяемым переменным, значения которых вычисляются, как правило, в результате итерационного процесса.

Сдвиг управляющей переменной цикла. Суть этой процедуры заключается в сдвиге на константу области изменения управля-

ющей переменной цикла. Например, участок (б) является результатом выполнения рассматриваемой процедуры над участком (а):

```
а) DOI = A BY N TO M;
   A (I) = B (I) * C (I);
   END;

б) DOI = A + N1 BY N TO M + N1;
   J = I — N1;
   A (J) = B (J) * C (J);
   END;
   I = J;
```

Существует другой вариант преобразований, выполняемых рассматриваемой процедурой. Участок (в) также является результатом выполнения процедуры сдвига управляющей переменной цикла над участком (а):

```
в) DOI = A + N1 BY N TO M + N1;
   A (I — N1) = B (I — N1) * C (I — N1);
   END;
   I = I — N1;
```

Процедура сдвига управляющей переменной цикла выполняется, как правило, перед процедурой объединения циклов, когда объединяемые циклы имеют различные области изменения управляющих переменных. Процедура сдвига циклов является частным случаем рассматриваемой процедуры.

Постфиксация операторов относительно ветвления. Если в программе перед оператором условного перехода, т. е. перед точкой ветвления, находится оператор присваивания, то с помощью рассматриваемой процедуры оператор присваивания перемещается за оператор условного перехода, т. е. копии оператора присваивания размещаются на каждой из ветвей, определяемых оператором условного перехода. Например, участок (б) является результатом выполнения процедуры постфиксации операторов относительно ветвления над участком (а):

```
а) 1  R = F
   2  IF (A. GE. O) GOTO 5
   3  E1 = A + 5.
   4  E2 = B + 5.
   5  GOTO 6
  65  D1 = A + 5.
   7  D2 = B + 5.
  86  C = E1 * D1 + E2 * D2
```

```
б) 2   IF (A. GE. O) GOTO 5
   1′  R = F
   3   E1 = A + 5.
   4   E2 = B + 5.
   5   GOTO 6
  1″5  R = F
   6   D1 = A + 5.
   7   D2 = B + 5.
  86   C = E1 * D1 + E2 * D2
```

На рис. 20, *а* и *б* показаны графы, соответствующие участкам (а) и (б) соответственно. Операторы $a_{1'}$ и $a_{1''}$ являются копиями оператора a_1. Рассматриваемая процедура выполняется перед

процедурой устранения неиспользованных определений, так как после размещения копий оператора на каждой из ветвей некоторые из этих копий могут оказаться неиспользованными определениями.

Префиксация операторов относительно слияния. Если в программе за меткой, на которую осуществляется несколько передач управления, т. е. за точкой слияния нескольких ветвей, находит-

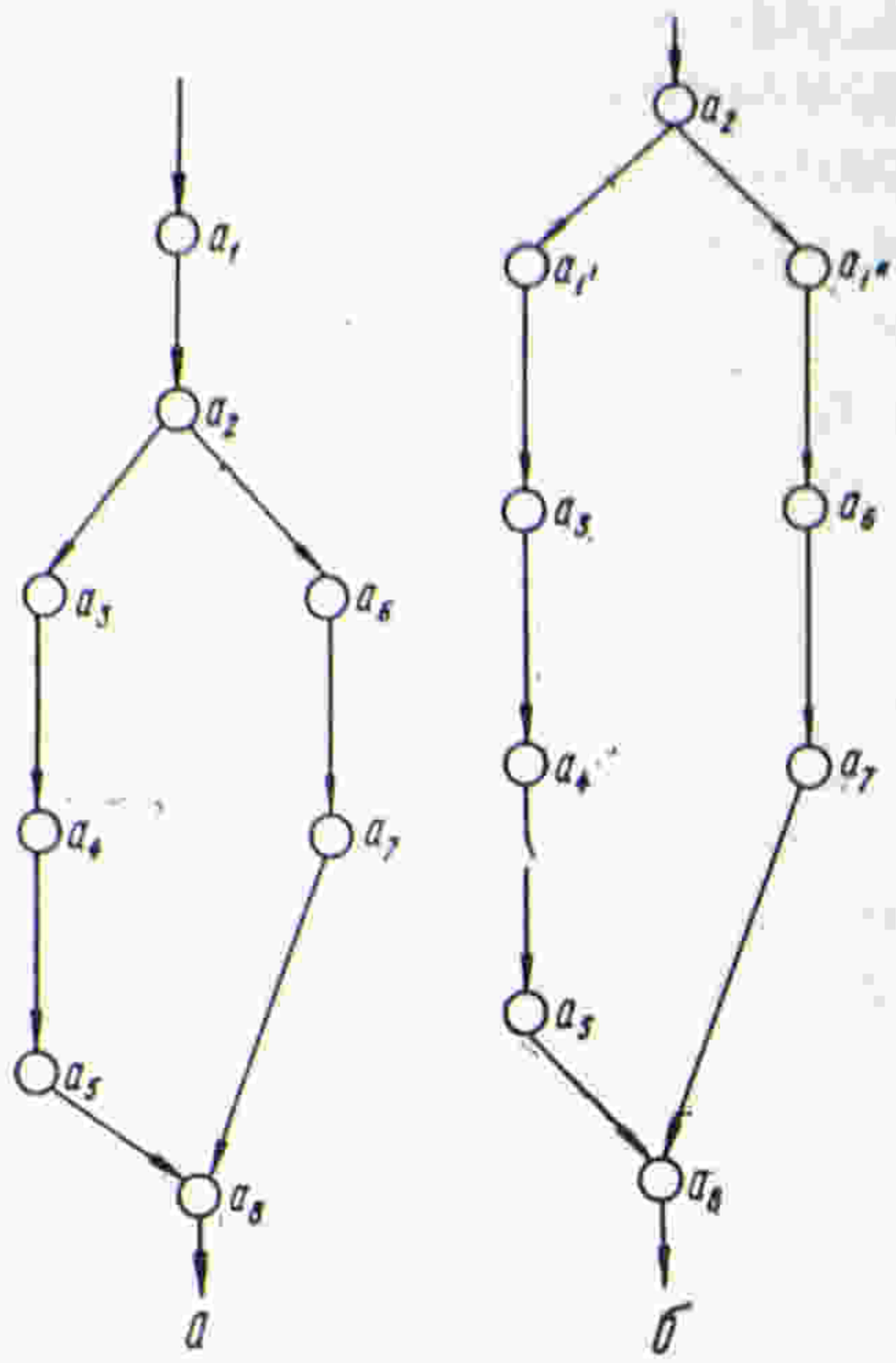


Рис. 20

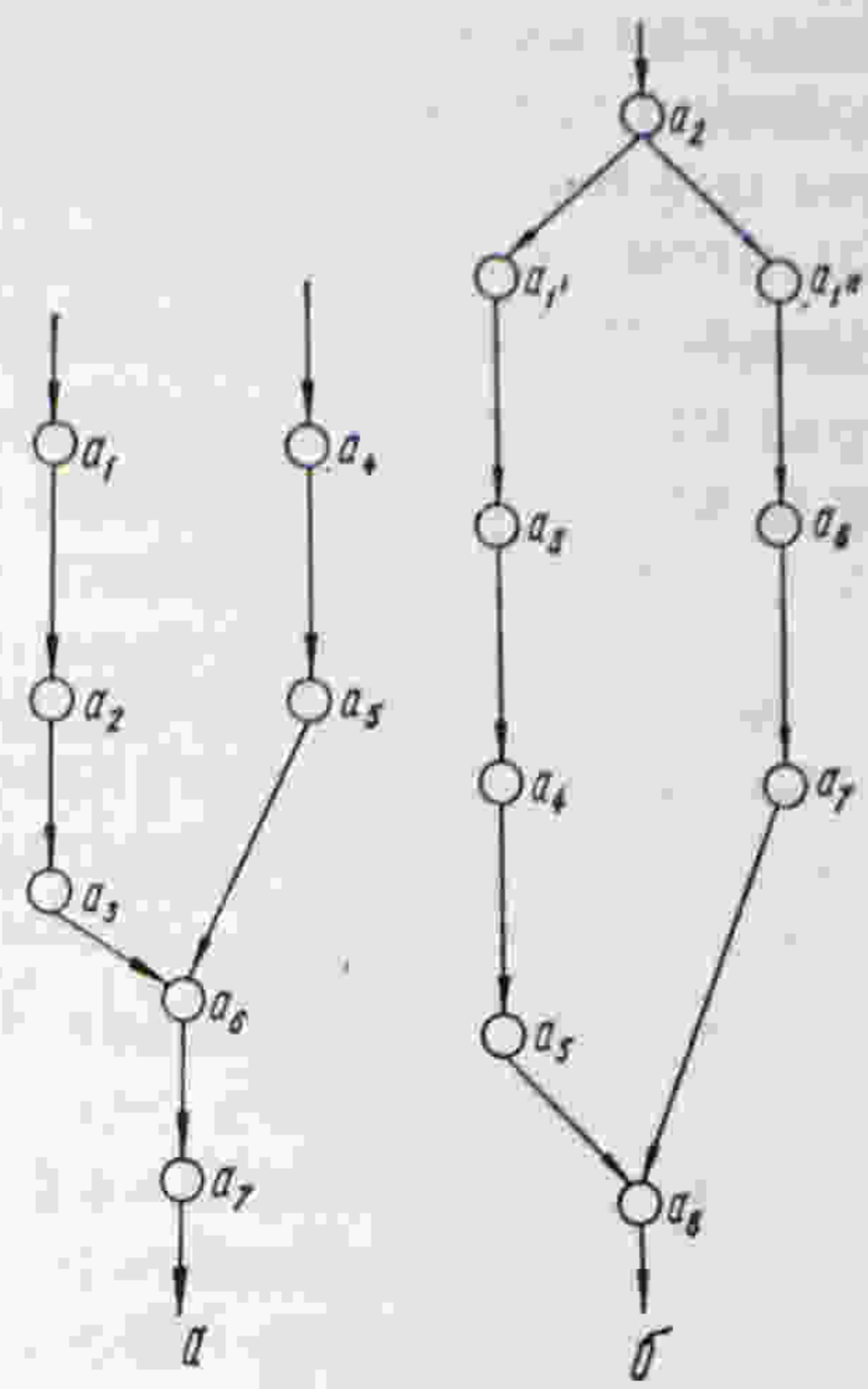


Рис. 21

ся оператор присваивания, то рассматриваемая процедура размещает оператор присваивания до точки слияния, т. е. размещает копии оператора присваивания на каждой из ветвей. Например, участок (б) является результатом выполнения процедуры префиксации операторов относительно слияния над участком (а):

```
а) 1     C1 = A ** 2        б) 1     C1 = A ** 2
   2     C2 = B ** 2           2     C2 = B ** 2
   3     GOTO 7                6'    C = C1/C2
   4 6   C1 = B ** 2           3     GOTO 7
   5     C2 = A ** 2           4 6   C1 = B ** 2
   6 7   C = C1/C2             5     C2 = A ** 2
   7     R = B + D             6''   C = C1/C2
                               7 7   R = B + D
```

На рис. 21, *а* и *б* показаны графы, соответствующие участкам (а) и (б) соответственно. Операторы $a_{6'}$ и $a_{6''}$ являются копиями оператора a_6. Рассматриваемая процедура, как и процедура постфиксации операторов относительно ветвления, выполняется перед процедурой устранения неиспользованных определений, так как после размещения копий оператора на каждой из ветвей некоторые

из этих копий могут оказаться неиспользованными определениями.

Переупорядочение операторов присваивания. Эта вспомогательная процедура предшествует в некоторых случаях и позволяет повысить оптимизационный эффект процедуры оптимизации выражений. Суть рассматриваемой процедуры заключается в переупорядочении операторов присваивания в пределах линейного участка или всей программы таким образом, чтобы результаты вычисления одних операторов присваивания могли быть использованы при вычислении других. Например, участок (б) является результатом выполнения процедуры переупорядочения операторов присваивания над участком (а):

```
а) A1 = SIN (X) ** 2 + B/E * 5.
   A2 = SIN (X) ** 2
   A3 = B/E * 5.
```

```
б) A2 = SIN (X) ** 2
   A3 = B/E * 5.
   A1 = SIN (X) ** 2 + B/E * 5.
```

Косвенный оптимизационный эффект, получаемый в результате выполнения рассматриваемой процедуры, можно оценить, в частности, сравнивая участки (в) и (г), являющиеся результатом выполнения процедуры оптимизации выражений над участками (а) и (б) соответственно (Y1, Y2 — вводимые в программу дополнительные переменные):

```
в) Y1 = SIN (X) ** 2
   Y2 = B/E * 5.
   A1 = Y1 + Y2
   A2 = Y1
   A3 = Y2
```

```
г) A2 = SIN (X) ** 2
   A3 = B/E * 5.
   A1 = A2 + A3
```

В зависимости от алгоритмов реализации процедура переупорядочения операторов присваивания может выполняться и после процедуры оптимизации выражений, однако в этом случае требуется выполнение дополнительных преобразований.

ГЛАВА 3

ПРАКТИЧЕСКИЕ АСПЕКТЫ ОПТИМИЗАЦИИ ПРОГРАММ

1. МЕТОДЫ ОПТИМИЗАЦИИ ПРОГРАММ

В настоящее время достаточно ясно, что оптимизация — это сложный эвристический процесс, в основе которого лежит общая стратегия, дополняемая множеством специальных методов, зависящих от класса оптимизируемых программ, различных человеческих и машинных ресурсов.

«Смешанная» стратегия. Область применения определенной оптимизационной процедуры и алгоритм ее реализации часто определяются контекстными условиями конкретной программы,

либо предыдущим этапом оптимизации. Например, если все формальные параметры некоторой подпрограммы используются в теле только этой подпрограммы, то при выполнении процедуры замены параметров локальными переменными все формальные параметры принадлежат первой группе, и алгоритм реализации этой процедуры упрощается. Аналогично, после выполнения процедуры удаления излишних операторов присваивания процедура выполнения константных действий применяется только к объединенному оператору. Разнообразие конструкций в реальных программах, использование их, как правило, в более узком смысле, чем тот, в котором они могут быть применены в общем случае, приводит к возникновению двух следующих ситуаций:

1) при выполнении оптимизационных процедур, учитывающих самые общие констекстные условия, время оптимизации значительно увеличивается за счет того, что простые контекстные зависимости обрабатываются достаточно сложным образом;

2) при выполнении оптимизационных процедур, учитывающих только одно простое контекстное условие, оптимизационный эффект этих процедур снижается, так как при более сложных контекстных условиях большинство неоптимальных конструкций не устраняется.

Сохранить приемлемое время оптимизации и получить достаточный оптимизационный эффект можно с помощью «смешанной» стратегии, широко используемой при организации эффективных процессов трансляции. Суть этой стратегии заключается в том, что универсальный алгоритм некоторого процесса (в частности трансляции) дополняется серией алгоритмов, рассчитанных на более узкие, но часто встречающиеся условия (контекст). В качестве успешного применения «смешанной» стратегии можно привести АЛЬФА-систему [2], где имеется, например, несколько способов трансляции нерекурсивных процедур.

Наиболее полный алгоритм некоторой оптимизационной процедуры, рассчитанный на максимально возможную область оптимизации и учет всех возможных управляющих и информационных связей, дополняется рядом более простых дополнительных алгоритмов этой процедуры, рассчитанных на применение в конкретном заранее известном контексте. Организация процесса оптимизации на основе «смешанной» стратегии наиболее целесообразна в следующих двух случаях.

1. Применение дополнительных упрощенных алгоритмов оправдано тогда, когда известно, что выполнение оптимизационной процедуры A влечет за собой выполнение оптимизационной процедуры B. В этом случае, как правило, контекстные условия выполнения процедуры B заранее известны. В частности, известны область оптимизации и некоторые управляющие и информационные связи. Если, например, процедура A повторна по отношению к процедуре B, то после выполнения A, как правило,

можно использовать упрощенный алгоритм реализации процедуры B. Таким образом, применение «смешанной» стратегии позволяет во многих случаях на основе учета контекста использовать более простые алгоритмы, чем те, которые рассчитаны на самый общий случай, что позволяет сократить время оптимизации программ без снижения оптимизационного эффекта выполняемых преобразований.

2. Применение «смешанной» стратегии оправдано в случае присутствия в исходной программе неоптимальных конструкций различной сложности, во-первых, редко встречающихся сложных неоптимальных конструкций, преобразование которых позволяет получить существенный оптимизационный эффект (выполнение таких преобразований требует значительных затрат ресурсов ЭВМ), во-вторых, аналогичных по типу часто встречающихся неоптимальных конструкций, преобразование которых позволяет получить меньший оптимизационный эффект с малыми затратами ресурсов ЭВМ. В таких условиях применение «смешанной» стратегии позволяет после анализа контекстных условий, т. е. потенциальной сложности преобразований, выбрать соответствующий алгоритм оптимизации.

«Смешанную» стратегию применяют также при наличии нескольких алгоритмов различной сложности реализации одной и той же процедуры, позволяющих в ряде случаев находить необходимый компромисс между временем трансляции (включающим время оптимизации) и степенью оптимизации программы, что является дополнительным удобством для пользователя, а в некоторых специальных случаях необходимым требованием для организации процесса оптимизации.

При удачном и достаточно детальном разбиении контекстных условий на классы и реализации набора оптимизационных процедур для каждого из этих классов, можно, во-первых, достичь значительного эффекта в экономии времени выполнения оптимизации и, во-вторых, найти необходимый компромисс между временем выполнения оптимизации и степенью оптимизации программы.

Применение «смешанной» стратегии позволяет настраивать систему оптимизации на характеристики конкретной программы и в некотором смысле, в зависимости от этих характерстик, управлять процессом оптимизации. Другими словами, «смешанная» стратегия содержит элементы параметрической настройки системы оптимизации на характеристики преобразуемой программы и некоторые элементы управляемой (на основе этих характеристик) оптимизации.

Факторизация программ. Часто различные ограничения на размеры области оптимизации (число операторов), которые возникают при практической реализации оптимизационных процедур, не позволяют распространить действие этих процедур на

всю программу. В этом случае осуществляется факторизация программы. Первоначально на основе элементарной единицы — оператора в программе — выделяются структуры, являющиеся областями оптимизации конкретной процедуры. Затем после выполнения оптимизационных процедур над выделенными структурами в качестве элементарной единицы рассматриваются эти структуры и выделяются структуры следующего уровня, являющиеся областями оптимизации и т. д. При факторизации программа представляет собой иерархию вложенных однотипных структур. Такими структурами могут быть гамаки, интервалы или зоны. При анализе программы рассматривают укрупненную структуру, элементарными конструкциями которой являются структуры непосредственно в нее вложенные. Факторизация позволяет распространить процесс оптимизации на всю программу, даже если область оптимизации существенно меньше программы. Однако при факторизации оптимизационный эффект в общем случае меньше того теоретического оптимизационного эффекта, который можно получить, если рассматривать в качестве области оптимизации всю программу. Это связано с тем, что при факторизации разрываются и не учитываются некоторые управляющие и информационные связи между переменными и операторами программы. Поэтому оптимизацию всей программы с использованием факторизации назовем здесь псевдоглобальной оптимизацией. На практике в связи с ограниченностью ресурсов ЭВМ и с потенциальным возрастанием сложности вычислений при увеличении числа управляющих и информационных связей теоретический оптимизационный эффект для больших программ недостижим. Поэтому факторизация — это один из важных методов, повышающих реальный оптимизационный эффект.

Наиболее распространенной является факторизация, когда в качестве элементарной единицы рассматривается линейный участок. В таком случае программа представляется в виде ориентированного графа (графа программы), у которого вершинам соответствуют линейные участки, а дугам — переходы между этими линейными участками. При такой факторизации первоначально выполняется локальная оптимизация на линейных участках. После выполнения локальной оптимизации строится граф программы, у которого для каждой вершины указана некоторая информация об операторах и переменных соответствующего линейного участка, например информация об используемых и определяемых переменных. Затем над факторизованной программой выполняется псевдоглобальная оптимизация.

В настоящее время достаточно хорошо развиты методы оптимизации, ориентированные на факторизацию программ, когда в качестве элементарной единицы рассматривается интервал. Для большинства реальных программ их управляющие графы являются сводимыми (см. гл. 1.3), т. е. могут быть путем факто-

ризации сведены к графу, состоящему из одной вершины. Это позволяет выполнять псевдоглобальную оптимизацию для реальных программ практически неограниченного объема.

Метод «меченых рыбок». Эффективность оптимизации определяется не только оптимизационным эффектом процедур, но и частотой выполнения преобразуемых участков программы или, как принято говорить, «частотным профилем» программы (иногда говорят «временным профилем» программы). В большинстве программ одни участки выполняются значительно чаще, чем другие. Исследования большой выборки программ показали, что около половины времени выполнения каждой программы приходится менее чем на 4 % ее операторов [112]. Частота выполнения отдельных участков программы, а более точно отношение суммарного времени выполнения этого участка к общему времени выполнения всей программы, является важным параметром, определяющим эффективность всего процесса оптимизации, с учетом которого определяется его стратегия. Чем больше отношение суммарного времени выполнения отдельного участка к общему времени выполнения всей программы, тем эффективней оптимизация этого участка. Действительно, для таких участков применение процедур даже с небольшим оптимизационным эффектом (по времени) за счет многократности выполнения преобразуемых участков позволяет получить значительный выигрыш времени. Полученный выигрыш равен оптимизационному эффекту процедуры, умноженному на число выполнений преобразуемого участка Поэтому, чем чаще выполняется некоторый участок программы, тем целесообразней применить к нему наиболее полную и тщательную оптимизацию. Под участком здесь понимается любая область: линейный участок, зона, гамак, фрагмент или отдельный программный модуль. Получение «частотных профилей» программы является важной вспомогательной задачей, так как если не сделаны точные измерения, то программист не сможет определить время, расходуемое различными частями его программы. Наиболее часто выполняемые части программы (или, как иногда говорят, наиболее нагруженные части программы) выявляют с помощью системы автоматического построения «частотных профилей» программ. Такие системы могут оценивать число прохождений через некоторый участок программы либо суммарное время выполнения этого участка в течение выполнения всей программы

Для определения частоты выполнения отдельных участков программы пользуются методом «меченых рыбок», пришедшим в программирование из физики.

Суть метода [illegible] следующем классическом примере, который [illegible] имеется озеро, в котором плавают рыбки, и необходимо определить число этих рыбок. Для этого из озера вылавливают n_1 рыбок, помечают их и снова выпускают в озеро. Затем, после того как меченые рыбки полностью смешались с немеченными, вылавливают n_2 рыбок. Пусть

в этой партии имеется n_3 меченых рыбок. Тогда с некоторой вероятностью можно сделать вывод, что в озере находится $n_1 n_2/n_3$ рыбки. Точность этой оценки зависит от соотношения чисел n_1, n_2, n_3.

В программировании время выполнения отдельных участков программы или время выполнения подпрограммы с помощью этого метода можно определить следующим способом. Программа пускается на счет. Через равные промежутки времени программист прерывает счет и фиксирует ячейку, на которой произошло прерывание. Зная размещение программы в памяти, можно определить, какой участок программы выполняется. На основе данных о частоте прерываний программы на определенном участке и общего времени выполнения программы можно определить время выполнения каждого участка программы. Действительно, пусть имеется I участков программы: N_i — число прерываний на i-м участке; $i \in (1, I)$; T — общее время выполнения программы; тогда согласно методу «меченых рыбок» время выполнения i-го участка с некоторой точностью равно $\left(N_i / \sum_{i=1}^{I} N_i\right) T$. Промежуток времени, через который происходят прерывания, определяет точность получаемых результатов. Этот способ удобен тем, что программист сам планирует стратегию эксперимента (промежуток времени между прерываниями) и может в каждом конкретном случае находить компромисс между числом прерываний и получаемой точностью оценки.

Для сравнения процесса определения времени выполнения отдельных участков программы или времени выполнения подпрограмм с процессом вылавливания рыбок из озера числу N_i можно сопоставить число n_1; времени выполнения отдельного участка программы или подпрограммы — число n_3; сумме прерываний на всех участках $\sum_{i=1}^{I} N_i$ — число рыбок в озере и общему времени выполнения программы T — число n_2. Из сопоставления видно, что в программировании метод «меченых рыбок» используется в несколько модифицированном виде, а именно, число всех рыбок в озере известно, и требуется оценить число выловленных во второй раз меченых рыбок, зная суммарное число меченых и немеченых рыбок, выловленных во второй раз.

Процесс получения данных о частоте выполнения отдельных участков программы можно автоматизировать, поместив в каждую интересующую программиста область оптимизации, например зону, счетчик, фиксирующий число выполнений этого участка. По окончании выполнения программы значения счетчиков определяют частоту выполнения соответствующих областей.

Управляемая оптимизация. В настоящее время большинство систем оптимизации программ создается на основе транслятора со встроенной в него и жестко зафиксированной совокупностью

процедур. Последовательность выполнения этих процедур также фиксирована. Иначе говоря, общая стратегия оптимизации определяется один раз при разработке системы и практически отсутствует возможность управления процессом оптимизации программы. Такой подход к построению систем оптимизации не позволяет, в общем случае, получить максимальный оптимизационный эффект (относительно реализованного набора процедур) для каждой конкретной программы. Приемлемым решением в таких случаях, по мнению авторов, является создание автоматизированных систем оптимизации программ с возможностью управления процессом оптимизации со стороны программиста, который может вмешиваться в этот процесс, изменять свою стратегию. Такой подход, по всей видимости, позволяет получать максимальный (в практическом смысле) оптимизационный эффект, если не для всех программ, то для определенных их классов.

Оптимизация (так же, как и отладка) программ — сложный эвристический процесс. Приступая к задаче оптимизации, программист не знает точного алгоритма ее решения, перебор же всех возможных вариантов (например, всех последовательностей применения оптимизационных процедур) практически неосуществим. Задачи такого рода целесообразно решать в режиме диалога человек — ЭВМ [9, 23, 78]. Общая схема цикла диалогового режима следующая: решающий предлагает машине просчитать некоторый вариант, анализирует полученный результат и на основе этого анализа предлагает следующий вариант. Эффективность диалогового режима зависит от опыта программиста и полноты той информации, которую программист получает для анализа.

При выполнении оптимизации управляющими параметрами программиста являются: набор применяемых оптимизационных процедур; последовательность применения этих процедур; вспомогательные алгоритмы, такие, как факторизация, получение «частотного профиля» программы и др.

Рассмотрим факторы, не позволяющие полностью автоматизировать процесс оптимизации (т. е. требующие участия программиста).

1. Выполнение ряда оптимизационных процедур может привести при некоторых наборах начальных значений к различным видам нарушений равносильности преобразований. Однако такие нарушения не всегда отражаются на функционировании программы. Формализация степени нарушения равносильности преобразований весьма затруднительна, так как даже факторы, влияющие на этот процесс, еще недостаточно изучены. Выполнение оптимизационных процедур такого типа возможно только под контролем программиста, когда он сам определяет возможность их включения. При этом выполнение процедур может распадаться

на две части. Первая часть выполняется автоматически, причем только для тех наборов начальных значений, которые гарантируют сохранение равносильности преобразований. Вторая часть — не поддающаяся формализации и, следовательно, автоматизации,— выполняется в режиме диалога для тех наборов начальных значений, для которых возможны нарушения равносильности выполняемых преобразований.

2. Критерий оптимизации в ряде случаев может быть достаточно сложной функцией от ряда параметров: времени выполнения, объема занимаемой памяти, частоты выполнения отдельных участков, сложности преобразуемой программы и т. д. В некоторых случаях, особенно для специальных программ (например, программ реального времени [77]), критерий оптимизации может быть просто неформализуемым. В этом случае либо невозможно автоматизировать оценку получаемого в результате выполнения отдельных процедур оптимизационного эффекта, либо это очень трудоемкая и дорогостоящая задача. Такую оценку в настоящее время, как правило, делают сами разработчики на основе интуитивных соображений.

3. Два критерия оптимизации (объем памяти и время выполнения программы), как правило, противоречат друг другу. Оптимизация программы по одному из критериев, в общем случае, влечет ее ухудшение по другому. Специальное математическое обеспечение ряда систем, например систем реального времени, содержит как фоновые программы, т. е. выполняемые по низшему приоритету, так и срочные, выполняемые в критических ситуациях по самому высокому приоритету. Если для фоновых программ критерием оптимизации, как правило, является объем памяти, то для срочных — это время их выполнения. Имеется целый ряд программ, рациональный критерий оптимизации которых находится между этими двумя крайними случаями. Для таких программ возникает задача оптимизации с граничными условиями, т. е. оптимизируя программу по времени выполнения, необходимо остаться в рамках заданного объема памяти и наоборот: при оптимизации по объему памяти время выполнения программы не должно превышать определенной величины. Автоматизация решения задачи оптимизации с граничными условиями либо требует больших затрат ресурсов ЭВМ, либо практически нереализуема.

Задачи оптимизации с неформализуемыми критериями оптимизации и с граничными условиями решаются путем многократного применения оптимизационных процедур. После каждого выполнения процедуры выполняется оценка полученных результатов, и на основании этой информации принимается решение о стратегии последующей оптимизации. Кроме того, имеются процедуры, которые осуществляют оптимизацию в несколько итераций; причем после каждой итерации на основе анализа получен-

ных результатов принимается решение о продолжении или прекращении выполнения данной процедуры. В качестве примера можно привести процедуру объединения циклов, где число объединяемых циклов ограничено предельным объемом результирующего цикла. При превышении этого объема эффективной становится противоположная ей процедура разделения циклов.

Однако режим диалога не характеризует получения максимального оптимизационного эффекта (относительно данного набора оптимизационных процедур), с его помощью можно лишь автоматизировать процесс поиска некоторой тупиковой программы, т. е. такой программы, которую не может преобразовать ни одна из включенных в набор оптимизационных процедур. Тупиковая программа может быть далека от совершенства и даже может не удовлетворять программиста по своему качеству. Применение режима диалога человек — ЭВМ в этом случае позволит, меняя последовательность оптимизационных процедур, находить тупиковую программу с приемлемыми характеристиками. В таком режиме программист определяет размеры областей оптимизации, структуры, являющиеся элементарными единицами при факторизации, и порядок выполнения оптимизационных процедур; ЭВМ производит преобразование программ на основе указаний программиста и выдает данные о преобразованной программе. Эти данные могут содержать различную информацию, индивидуальную для каждой конкретной задачи и необходимую для проведения дальнейшей оптимизации. Программист на основе полученных данных в зависимости от целей оптимизации принимает решение о дальнейшей стратегии.

При построении автоматизированных систем особое значение имеет изучение взаимозависимостей оптимизационных процедур, их тупиковых и повторных свойств. В настоящее время теоретические аспекты этого вопроса еще недостаточно исследованы, эффективность процесса диалога человек — ЭВМ во многом зависит от содержания информации, получаемой пользователем. Значения двух параметров — времени счета и объема памяти, занимаемой программой, до и после выполнения оптимизационных процедур — явно недостаточны как информация для ведения эффективного диалога. Здесь, по мнению авторов, большое значение имеет создание специальных инструментариев, обеспечивающих программиста информацией, необходимой для выполнения оптимизации, и методами доступа к этой информации. Такой информацией могут быть управляющий и информационный графы программы; их отдельные фрагменты; информация о частоте выполнения определенных участков программы; области значений переменных [3]; объектный код, соответствующий отдельным фрагментам программы, и т. д. Создание такого инструментария в сочетании с принципами управляемой оптимизации позволит качественно изменить процесс создания, который тогда можно будет

называть процессом проектирования (конструирования) оптимальной программы.

Основой для создания систем диалоговой управляемой оптимизации служат процедуры различного значения, с разными областями действия, эффективными в различных контекстных условиях, и т. д.; т. е. процедуры того набора, который уже имеется в системах со «смешанной» стратегией.

Кроме получения достижимого оптимизационного эффекта, диалоговая управляемая оптимизация позволяет: отключать оптимизационные процедуры, выполнение которых приводит к нарушению равносильности преобразований или к отрицательному оптимизационному эффекту для конкретной программы; а также прекращать процесс оптимизации при недостатке памяти ЭВМ или слишком большом времени оптимизации. Следует также заметить, что управляемая оптимизация не есть альтернатива оптимизации с жестко зафиксированной последовательностью выполняемых процедур. Очевидно, в будущем на практике наиболее эффективными будут системы оптимизации, которые по требованию программиста могут работать в режиме «смешанной» стратегии или диалоговой управляемой оптимизации.

Конкретизация программ. В современном программировании большое значение имеет создание пакетов прикладных программ (ППП), ориентированных на широкий круг пользователей. Такие ППП, как правило, обладают значительной универсальностью, а пользователь использует их в некотором узком классе задач. За счет этого возникает несоответствие между тем широким набором функций, которые может реализовать ППП, и теми несколькими функциями, которые реализуются в действительности. Такое несоответствие в каждом конкретном случае предоставляет дополнительные возможности для оптимизации как всего ППП, так и отдельных его программ, за счет того, что универсализация ухудшает качество программ. Оптимизация осуществляется на основе учета конкретных взаимосвязей между отдельными модулями ППП, способов использования этих модулей, информации о начальных значениях переменных, о соотношениях между переменными (например, $x < y$) и типах структур (например, матрица A является диагональной), а также на основе информации об используемой операционной системе и ЭВМ, на которой выполняется программа. Такая оптимизация называется конкретизацией программы. В частности, некоторые оптимизационные процедуры, ориентированные на привязку программы к конкретному контексту, выполняют конкретизацию программы.

Возможности улучшения программы путем ее конкретизации рассмотрим на примере процедуры MULT [показанной на участке (а)], осуществляющей умножение двух матриц: матрицы X размерности $L \times M$ и матрицы Y размерности $M \times N$, результатом умножения является матрица Z размерности

L × N (используется язык ПЛ/1):

```
а) MULT : PROCEDURE (X, Y, Z, L, M, N);
   DECLARE (X (L, M), Y (M, N), Z (L, N)) FIXED (15. 3),
   (L, M, N) FIXED (3. 0);
   DO I = 1 TOL;
   DO J = 1 TON;
   Z (I, J) = 0;
   DO K = 1 TOM;
   Z (I, J) = X (I, K) * Y (K, J) + Z (I, J);
   END;
   END;
   END;
   END MULT;
```

Основой для конкретизации является информация о значениях размеров массивов: L = 20; M = 20; N = 30, а также то, что матрица X является диагональной. Информацию о том, что матрица X является диагональной, можно формализовать следующим образом:

```
IF I = K THEN X (I, K) = X (I, K);
ELSE X (I, K) = 0;
```

После выполнения процедуры свертки формальных параметров участок (а) принимает следующий вид:

```
б) MULT : PROCEDURE (X, Y, Z);
   DECLARE (X (20, 20), Y (20, 30), Z (20, 30)) FIXED (15. 3);
   DO I = 1 TO 20;
   DO J = 1 TO 30;
   Z (I, J) = 0;
   DO K = 1 TO 20;
   Z (I, J) = X (I, K) * Y (K, J) + Z (I, J);
   END;
   END;
   END;
   END MULT;
```

Конкретизируем процедуру участка (а) на основе информации о диагональности матрицы X, используя формальное представление этой информации:

```
в) MULT : PROCEDURE (X, Y, Z);
   DECLARE (X (20, 20), Y (20, 30), Z (20, 30)) FIXED (15. 3);
   DO I = 1 TO 20;
   DO J = 1 TO 30;
   Z (I, J) = 0;
   DO K = 1 TO 20;
   IF I = K THEN
   Z (I, J) = X (I, K) * Y (K, J) + Z (I, J);
   ELSE
   Z (I, J) = O * Y (K, J) + Z (I, J);
   END;
   END;
   END;
   END MULT;
```

После применения процедуры выполнения аналитических преобразований, а затем процедуры удаления тождественных операторов участок (а) принимает следующий вид:

```
г) MULT : PROCEDURE (X, Y, Z);
   DECLARE (X (20, 20), Y (20, 30), Z (20, 30)), FIXED (15. 3);
   DO I = 1 TO 20;
```

```
DO J = 1 TO 30;
Z (I, J) = 0;
DO K = 1 TO 20;
IF I = K THEN Z (I, J) = X (I, K) * Y (K, J) + Z (I, J);
END;
END;
END;
END MULT;
```

Выполнение процедуры устранения циклов, а затем процедуры подстановки позволяет преобразовать участок (г) в участок (д):

```
д) MULT : PROCEDURE (X, Y, Z);
   DECLARE (X (20, 20), Y (20, 30), Z (20, 30)) FIXED (15. 3);
   DO I = 1 TO 20;
   DO J = 1 TO 30;
   Z (I, J) = 0;
   IF1 ≤ 20 & 1 ≤ 20 & MAX (1, 1) ≤ I & I ≤ MIN (20, 20)
   THEN Z (I, J) = X (I, I) * Y (I, J) + Z (I, J);
   END;
   END;
   END MULT;
```

Так как область изменения управляющей переменной I лежит в пределах от 1 до 20, то после выполнения процедуры ликвидации константных распознавателей участок (д) принимает следующий вид:

```
е) MULT : PROCEDURE (X, Y, Z);
   DECLARE (X (20, 20), Y (20, 30), Z (20, 30)) FIXED (15. 3);
   DO I = 1 TO 20;
   DO J = 1 TO 30;
   Z (I, J) = 0;
   Z (I, J) = X (I, I) * Y (I, J) + Z (I, J);
   END;
   END;
   END MULT;
```

К участку (е) применим процедуры подстановки и выполнения аналитических преобразований:

```
з) MULT : PROCEDURE (X, Y, Z);
   DECLARE (X (20, 20), Y (20, 30), Z (20, 30)) FIXED (15. 3);
   DO I = 1 TO 20;
   DO J = 1 TO 30;
   Z (I, J) = X (I, I) * Y (I, J);
   END;
   END;
   END MULT;
```

На этом процесс конкретизации рассматриваемой программы заканчивается. Полученная программа участка (з) является существенно более эффективной, чем эквивалентная ей исходная программа участка (а).

Таким образом, чем больше имеется сведений о свойствах данных, свойствах программы, особенностях ее выполнения, тем больше возможностей для конкретизации программы. Такую исходную информацию для конкретизации программы можно получить или от пользователя путем прямых указаний, или в результате автоматического анализа программы и взаимосвязей между переменными и операторами.

На приведенном примере конкретизация выполнена традиционными методами оптимизации — применением к программе последовательности оптимизационных преобразований. В последние годы разработан другой подход к конкретизации программ, основанный на понятии смешанного вычисления [41, 42, 58]. Под смешанным вычислением для фиксированного языка программирования понимается некоторый алгоритм, результатом выполнения которого является остаточная программа, получаемая из исходной связыванием ее аргументов. Смешанные вычисления реализуются как применение системы преобразований, называемых базовыми трансформациями.

Выполнение базовых трансформаций над исходной программой $П$ с исходным состоянием памяти X приводит к остаточной программе T и промежуточному состоянию памяти Y таким, что результат применения $П$ к X совпадает с результатом применения T к Y, т. е. смешанные вычисления являются настройкой программы на поток конкретных данных.

Конкретизация программы является, по своей сути, как и смешанные вычисления, частичным или полным выполнением некоторых операторов программы в процессе ее трансляции.

Следует отметить, что оптимизация программ посредством их конкретизации приобретает в настоящее время особое значение, так как в связи с накоплением большого числа программных модулей одним из развивающихся направлений программирования является составление программ из различных уже готовых модулей. Такой метод составления программ приводит к необходимости изменения этих модулей и их настройки на конкретные контекстные условия, в частности, на поток конкретных данных. Из модулей, разработанных в разное время (на различных этапах развития программирования) программистами различной квалификации трудно составить программу (тем более эффективную) без предварительной конкретизации этих модулей, даже если они содержат все функции для решения требуемой задачи.

2. ЭТАПЫ ТЕХНОЛОГИИ ПРОИЗВОДСТВА ПРОГРАММ

Процессу собственно оптимизации программы предшествуют процессы анализа управляющего и информационного графов программы и сбора необходимой информации. Эта информация должна быть достаточной для того, чтобы можно было моделировать процесс выполнения программы для всего множества (либо для достаточно широкого подмножества) начальных данных. Кроме того, на основании этой информации необходимо судить об изменениях значений переменных, об их взаимозависимостях, об очередности выполнения операторов и о различных свойствах участков программ. Аналогичные задачи возникают и в ряде других областей программирования. Все это позволяет исполь-

зовать информацию, полученную на этапе анализа управляющего и информационного графов, не только для оптимизации программ, но и для других целей на различных этапах технологического процесса создания программы, в частности, на этапах трансляции, отладки и верификации, модификации, документирования и сопровождения программы [59].

Трансляция — это процесс перевода программы с одного языка на другой. Разделяют трансляцию программы на машинный язык (или близкий к нему) и на другой язык, но того же (или близкого) уровня. Роль оптимизации программ в процессе трансляции первого вида общеизвестна и хорошо исследована. Значительный интерес представляет использование методов оптимизации для трансляции второго вида. Методы оптимизации для такого вида трансляции могут быть использованы как в традиционном плане (для устранения избыточных вычислений; преобразования структуры программы к более эффективному виду и т. д.), так и в плане перевода программы с одного языка на другой. В этом случае информация, собранная на этапе анализа управляющего и информационного графов программы, служит для организации процесса перевода.

Предварительный анализ управляющего и информационного графов программы и последующая ее оптимизация позволяют создавать трансляторы, в основе которых лежат заведомо упрощенные алгоритмы трансляции, учитывающие выполненную оптимизацию, т. е. трансляторы, производящие сравнительно простую генерацию машинного кода, когда для компиляции машинной команды требуется анализ одного или, может быть, только части одного оператора входного языка, а это является дополнительной предпосылкой для создания эффективной технологии автоматического или полуавтоматического производства трансляторов.

Отладка и верификация. Под отладкой понимают поиск и устранение синтаксических и семантических ошибок, а под верификацией доказательство того, что программа реализует заданную функцию. Методы отладки (а именно, поиска семантических ошибок) и верификации программ зачастую совпадают, и сами эти задачи по своей сущности являются достаточно близкими, а в некоторых своих аспектах взаимозаменяемыми. Более того, отладку и верификацию можно объединить одним термином — проверкой правильности программ, где частичную проверку правильности программ осуществляет отладка и полную проверку — верификация. Отладка и верификация, как правило, сводятся к проверке правильности взаимозависимостей логических величин, правильности вычисления переменных, контролю взаимодействия участков программы по управлению. При этом проверяют основные маршруты преобразования информации, определяют все управляющие связи и связи по информации на входе

и выходе подпрограмм — отдельных модулей и программы в целом.

Степень отлаженности программы зависит от совокупности тестов, каждый из которых соответствует некоторому набору начальных значений отлаживаемой программы. При отладке программы стараются покрыть все переходы и пройти все пути программы. Традиционно эту задачу пытаются решить подбором множества тестовых начальных данных. Идея автоматической верификации программ основана на различного рода утверждениях, содержащих достаточно глубокую информацию об управляющих и информационных зависимостях переменных, операторов и участков программы.

Методы анализа управляющего и информационного графов, используемые при решении задачи оптимизации программ, позволяют собрать информацию о программе, которую можно использовать для решения этих задач. Более того, алгоритмы ряда оптимизационных процедур можно практически без изменений применить для решения задач отладки и верификации, например для обнаружения переменных, которые используются не будучи предварительно определены; для определения областей действий переменных; для трассировки маршрутов переменных; для обнаружения избыточных операторов печати, используемых в процессе отладки; для трассировки программ и т. д. Следует отметить, что задачи отладки и верификации, так же, как и задача оптимизации программ, алгоритмически неразрешимы. Поэтому такие хорошо разработанные и апробированные специальные приемы и методы, как выделение специальных областей в программе; параметрическая настройка на входной язык (см. гл. 3, 4); «смешанная» стратегия; факторизация программ; метод «меченых рыбок»; стратегия работы в диалоговом режиме; конкретизация программы, используемые при решении задачи оптимизации программ, могут быть использованы для решения задач отладки и верификации.

В качестве примера оптимизирующего транслятора, у которого на основе методов оптимизации реализован ряд средств отладки программ, можно привести транслятор FOREX [98], который помимо традиционной диагностики, выдает также сообщения о всех переменных, значения которых не определены, но используются, или наоборот, определены, но не используются, и другую информацию, полученную при анализе управляющего и информационного графов.

По мнению авторов, методы отладки и оптимизации в будущем будут все более тесно связаны между собой, так как можно выделить устойчиво содержащиеся в программах ошибки (так же, как это сделано при выделении основных типов неоптимальных конструкций), а стратегия целенаправленного поиска таких ошибок близка стратегии поиска неоптимальных конструк-

ций. Попытки выделения устойчивых типов ошибок предпринимались [53, 70]. Учитывая результаты этих работ и опыт практической отладки программ [13], авторы выделяют следующие устойчиво содержащиеся в программах ошибки: неправильный тип, структура или значение константы; неправильный сдвиг регистра; неверное выравнивание переменных; наличие знаменателей, способных обратиться в нуль; ошибочное применение одних синтаксических конструкций вместо других; ошибки перфорирования и фиксирования программы в памяти; неверная спецификация или формат данных; неправильные употребление вложенных операторов IF, формирование и употребление логических выражений; результат арифметических вычислений, обусловленный спецификой процесса программирования; неверная передача управления оператором перехода; отсутствие анализа кодов ответа; неверный адрес в дискрипторе; ошибки в работе с магазинной памятью; обращение с константами как с параметрами; непредусмотренное изменение модулей входных параметров; неправильный тип или значение фактического параметра процедуры; сдвиг на единицу при индексировании; бесконечное повторение одной и той же последовательности операторов; ошибки в организации циклов, при вводе-выводе информации; использование неправильной версии программы; ошибки нарушения стандартов программирования, отклонения от спецификаций; переход внутрь цикла; наличие циклов, операторов, которые никогда не могут быть выполнены; переход в область данных программы; незавершенность вычислительного процесса; отсутствие установки рабочих регистров, обнуления ячеек памяти ОЗУ, начальной установки переменных; ошибки начальных установок, использования малоразрядной информации; счетчики слишком малой разрядности; наложение массивов или переменных; неправильная расстановка и функционирование флажков; функциональное несоответствие программы алгоритму; мерцающие ошибки (появляющиеся время от времени); отсутствие ограничителя (разделителя); непредусмотренное сокращение или расширение маршрутов обработки информации; ошибочная обработка прерываний; не предусмотрены особые ситуации; неправильное окончание программных последовательностей; неверное использование переменных многими модулями; некорректное внесение изменений в программу; временные ошибки; ошибки во вводимых исходных данных.

Приводимый набор (не претендующий на полноту) содержит ошибки, которые осмыслены и хорошо поняты программистами. При определении этого набора учитывались программы, реализующие широкий спектр алгоритмов и написанные на языках разных уровней программистами различной квалификации для разных типов ЭВМ. В зависимости от характеристик рассматриваемого класса программ, а также требуемой надежности рассмат-

ривается некоторое подмножество ошибок. Наличие в программе устойчивых ошибок позволяет реализовать отладочные процедуры, каждая из которых исправляет соответствующую ошибку. Эти отладочные процедуры по своему содержанию аналогичны оптимизационным процедурам.

Рассмотренный подход к отладке программ, который можно назвать методом целенаправленного поиска конкретных ошибок, позволяет находить многие ошибки с меньшими затратами, чем традиционные методы тестирования программ. Авторы рассматривают предлагаемый подход не как альтернативный, а как дополнение (на этапе предварительной обработки программы) к традиционным методам тестирования. Такое сочетание позволит сократить расходы ресурсов при отладке программ без снижения их качества, так как в соответствии с рассматриваемым подходом, после предварительной отладки программы у разработчика есть уверенность в том, что определенные ошибки отсутствуют в программе. В связи с алгоритмической неразрешимостью общей задачи отладки программ при поиске конкретных ошибок весьма перспективна управляемая отладка, в частнсости, в режиме диалога человек — ЭВМ.

Между верификацией и оптимизацией, по-видимому, существует такая же взаимосвязь, как между отладкой и оптимизацией, хотя она в настоящее время меньше исследована. Таким образом, очевидно, можно выработать ряд стандартных приемов — верификационных процедур (аналогичных оптимизационным процедурам), применение которых к определенным участкам программ позволит доказать правильность или неправильность этих участков. В частности, в работе [36] описаны несколько приемов доказательства правильности участков программ (в данном случае циклов), рассматриваемых в качестве прообраза некоторых верификационных процедур. Выделение набора верификационных процедур позволяет строить системы верификации программ аналогично, так же как системы оптимизации программ, описанные в настоящей книге. В связи с алгоритмической неразрешимостью общей задачи верификации программ, так же, как и в системах оптимизации и системах отладки программ, весьма перспективна управляемая верификация, в частности, в режиме диалога человек — ЭВМ.

Модификация. Под модификацией понимается внесение изменений в уже отлаженную и правильно функционирующую программу. Такие изменения связаны в основном с изменением алгоритма решения задачи, а также с созданием любого достаточно сложного комплекса программ. Внесение даже простых изменений в программу — это далеко не тривиальная задача, так как в программе большого объема и сложной структуры они затрагивают большое число управляющих и информационных связей, что может вызвать появление нежелательных эффектов. Как пра-

вило, программист не в состоянии предвидеть все подобные эффекты и поэтому, для того чтобы быть уверенным в работоспособности модифицированной программы, повторно осуществляет процесс отладки и верификации модифицированной программы. Однако задача модификации программ существенно упрощается, если программисту будут автоматически сообщаться возможные эффекты необходимого изменения и связанные с данным изменением потенциальные места появления ошибок. Для такой автоматизации можно использовать технику, применяемую при оптимизации программ, а именно, методы и алгоритмы, осуществляющие глобальный анализ управляющих и информационных связей программы. Методы, используемые при оптимизации программ, позволяют более целенаправленно модифицировать программу за счет явного проявления изменений, вносимых в управляющий и информационный графы программы. Автоматизация процесса модификации программ (помимо своего прямого назначения) облегчит выполнение отладки и верификации программ, так как эти процессы тесно связаны между собой.

Опыт построения систем, являющихся прообразом систем модификации программ, позволяет говорить о некотором стандартном наборе модификационных процедур, аналогичных оптимизационным процедурам. Практически система модификации программ будет достаточно полной, если в ней реализовать набор модификационных процедур, аналогичных набору редактирующих директив. Отличие модификационных процедур от редактирующих директив состоит в том, что первые учитывают влияние на управляющий и информационный графы программы производимых в ней изменений, в то время как вторые просто производят такие изменения. Выделение набора модификационных процедур позволяет строить системы модификации программ так же, как системы оптимизации программ. Здесь также следует подчеркнуть, что в связи со сложностью управляющих и информационных связей программы, управляемая модификация, в частности, диалог человек — ЭВМ, имеет особое значение в системах модификации программ (так же, как и в системах оптимизации программ и в других рассмотренных системах).

Документирование. Получение хорошей документации является одной из важных задач процесса создания программного продукта. Можно определить следующие основные виды документации на алгоритмы и программы [71]: описание алгоритма; распределение памяти; описание памяти; описание перекрестных ссылок; структурная схема программы; листинг программы; документация для технической реализации программы.

Если описание алгоритма и получение документации для технической реализации программы выполняет разработчик программы, как правило, на естественном языке, то получение остальных документов можно автоматизировать, используя методы

оптимизации программ. Действительно, распределение и описание памяти можно получить на основе анализа информационного графа программы, а описание перекрестных ссылок — на основе анализа управляющего графа программы. Структурная схема — это по сути дела описание иерархии вложенности фрагментов программы и фиксации информационно-управляющих связей между ними. Выделение такой иерархии как раз и является одной из задач, решаемых в процессе анализа управляющего и информационного графов при оптимизации программ. Кроме того, техника оптимизации программ позволяет автоматизировать процесс обеспечения программы различными комментариями о взаимосвязях переменных и операторов, т. е. листингами, что необходимо для повышения читаемости и понимаемости программ.

В настоящее время процесс оформления документации и сама документация на программный продукт все больше подчиняются требованиям различных ГОСТов, т. е. происходит как стандартизация документации, так и стандартизация процесса ее изготовления. Это способствует выработке определенных приемов, которые можно назвать документационными процедурами. Результатом применения таких документационных процедур к тексту программы является изготовление определенных документов, содержащих информацию о программе. Для реализации документационных процедур необходима та же информация, такой же анализ управляющего и информационного графов, как и для оптимизационных процедур. Поэтому принципы создания систем документации программ, в основе которых лежит применение документационных процедур к тексту программы, во многом аналогичны принципам создания систем оптимизации программ.

Сопровождение программ. При сопровождении комплексов программ возникает задача устранения из программ ошибок двух типов: обусловленных помехами и сбоями аппаратуры и обнаруживаемых только в условиях промышленной эксплуатации системы. При наличии ошибок первого типа в простейшем случае программу просто копируют, однако для памяти с односторонним доступом решение этой задачи сводится к тестированию программы и после обнаружения ошибки к замене соответствующего блока памяти. Локализация и последующее исправление ошибок второго типа, как правило, также сводится к тестированию программы. Успех тестирования определяется системой тестов, к которым предъявляются требования быстрой и точной локализации таких ошибок. Использование для выбора тестов информации, получаемой в результате анализа управляющего и информационного графов программы методами оптимизации, упрощает процесс локализации таких ошибок. Эта информация позволяет выбрать тесты для прохождения запланированных

маршрутов, что, в свою очередь, позволяет по следствию — неправильному выполнению операторов в результате появления ошибки, найти причину-ошибку.

3. СТРУКТУРНЫЙ ПОДХОД К ПРОГРАММИРОВАНИЮ

Традиционные методы программирования складывались и долгое время оставались неизменными в условиях небольших вычислительных мощностей и программ небольшого объема, как правило, научного характера и одноразового использования. Проникновение вычислительной техники во все области науки и техники, в управление отраслями народного хозяйства существенно изменило сложившуюся ситуацию: резко возрос объем создаваемого программного продукта, повысились требования к его качеству и надежности. Методы, используемые для создания качественной и надежной программы в 50 операторов, совсем непригодны для создания аналогичной по содержанию и столь же качественной и надежной программы в 5000 операторов. Здесь проявляется тот общеизвестный факт, что любые два объекта, которые отличаются по объему и сложности в сто или более раз, становятся совершенно несопоставимыми. Кроме того, с ростом объема программы, начиная с некоторого уровня, сроки ее создания увеличиваются не в арифметической, а, в лучшем случае, в геометрической пропорции. В результате появилось в различных странах и в разных областях программирования много методов и приемов, направленных на создание программ большого объема в приемлемые сроки. Одним из таких методов создания (проектирования) программ, вобравшим в себя богатый опыт большого числа программистов, хорошо апробированным и, как следствие, получившим широкое признание, является структурный подход к программированию [21, 22, 25, 30, 31, 33, 36—39, 53, 92, 94, 95].

Структурный подход к программированию, как и оптимизация программ, имеет одну и ту же область приложения, а именно, программы большого объема *.

Нисходящая разработка. Проектирование программ. При нисходящей разработке разбиение программы на модули осуществляется эффективным образом, при этом полученные модули обладают определенными свойствами, из которых отметим только то, что каждый модуль должен реализовать только одну функцию, причем целиком. Принцип «один модуль — одна функция» рассматривается как ключ к хорошему проектированию программ. Благодаря этому принципу программа получается легко

* В связи с тем, что в литературе существуют расхождения относительно некоторых определений и терминологии структурного подхода к программированию, то при изложении основных принципов этого подхода авторы придерживались работы [92].

понимаемой с явно просматриваемыми связями между функциями. Нисходящее проектирование основано на идее уровней абстракций, которые соответствуют уровням модулей в создаваемой программе.

Характерной особенностью нисходящего проектирования (с учетом этого принципа) является то, что выделяемые модули становятся в совокупности максимально независимыми друг от друга: в смысле подчинения — каждый модуль подчиняется только вышестоящим над ними по иерархии модулям; в информационном плане — уменьшается объем пересылаемой между модулями информации. Такое формирование модулей можно рассматривать как факторизацию программы, причем факторизацию естественную, основанную на функциональных особенностях алгоритма. Здесь факторизация, в отличие от традиционной факторизации, применяемой при оптимизации, осуществляется сверху вниз, а не снизу вверх. В соответствии с нисходящим проектированием сначала строится самый укрупненный граф программы, вершинам которого соответствуют модули верхнего уровня, затем каждой вершине этого графа ставится в соответствие более подробный граф, отражающий разбиение соответствующего модуля на модули более низких уровней, и т. д. На этапе проектирования только выделяются вершины графов различных уровней, а управляющие и информационные связи между вершинами строятся на этапе записи программы на псевдокоде [38], т. е. на этапе ее кодирования. Преимущества рассматриваемой факторизации сверху вниз по сравнению с традиционной заключаются в следующем:

1. Факторизация не существует как отдельный дополнительный процесс, а по сути дела совпадает с процессом проектирования, являющимся необходимым и самостоятельным этапом создания программы, т. е. факторизация выполняется без дополнительных затрат как человеческих, так и машинных ресурсов, что удешевляет общий процесс оптимизации программ.

2. Следование принципу «один модуль — одна функция» приводит к уменьшению управляющих и информационных связей между вершинами управляющих и информационных графов различных уровней, получаемых в результате факторизации. Это обусловливает упрощение графов и, следовательно, снижение затрат на анализ этих графов и последующее их преобразование оптимизационными процедурами. Снижение таких затрат может повысить оптимизационный эффект выполняемых преобразований, так как зачастую повышение этого эффекта сдерживается имеющимися в наличии ресурсами ЭВМ.

3. Следование принципу «один модуль — одна функция» повышает наглядность получаемых в результате рассматриваемой факторизации управляющих и информационных графов различных уровней, так как эти графы отражают функциональную

(содержательную) структуру программ. При реализации управляемой оптимизации, когда управляющий и информационный графы доступны программисту, наглядность имеет существенное значение, так как позволяет точнее оценить особенности программы и, следовательно, выбрать более правильную стратегию ее дальнейшей оптимизации. Кроме того, легкая читаемость и понимаемость программ часто позволяют разработчику самому вовремя заметить неоптимальные участки и исправить их, что также способствует повышению эффективности процесса оптимизации в целом.

4. Следование принципу «один модуль — одна функция» уменьшает возможность неудачного разбиения программы на модули, т. е. структурный подход к программированию в минимальной степени подвержен влиянию этого «вредного» побочного эффекта модульного программирования.

Кроме того, сама по себе факторизация в рамках нисходящего проектирования, по мнению авторов, обладает в потенциале большими оптимизационными возможностями, чем традиционная факторизация по методу снизу вверх. В основе этого лежит тот же принцип, который лежит в основе повышения надежности программ при проектировании сверху вниз по сравнению с проектированием снизу вверх, а именно: сначала решать самые главные, глобальные задачи, а только затем переходить к решению более мелких задач. Заметим, что этот принцип несколько перекликается с оптимизацией программ на языках различных уровней, когда в процессе оптимизации происходит переход с языка высокого уровня на язык более низкого уровня. При таких переходах также сначала решаются глобальные задачи — преобразуются алгоритм и самые крупные конструкции программы, а затем более мелкие, вплоть до отдельных команд ассемблера. Иначе говоря, принцип последовательной оптимизации программ на языках различных уровней (с понижением уровня языка) является прообразом некоторых идей структурного подхода к программированию в приложении к оптимизации программ.

Планирование программирования. В рамках нисходящей разработки существуют три способа планирования программирования модулей: иерархический, операционный и их комбинация.

При иерархическом способе порядок программирования и тестирования модулей определяется их расположением в схеме иерархии. Сначала программируются и тестируются все модули одного уровня, затем происходит переход на уровень ниже и т. д. При операционном способе модули программируются в соответствии с порядком их выполнения при запуске готовой программы. Оба способа, иерархический и операционный, обладают своими преимуществами и недостатками. Третий способ, являющийся комбинацией двух рассмотренных, позволяет сохранить их преимущества и в то же время избавиться от некото-

рых их недостатков. Выбор способа планирования непосредственным образом связан с эффективностью процесса оптимизации программ. Дело в том, что каждый из этих способов определяет последовательность расположения текстов модулей на листинге, т. е. определяет читаемость и понимаемость программы. А эти факторы существенно влияют на эффективность процесса оптимизации, в частности, управляемой оптимизации.

Все три способа планирования и реализации программ в рамках нисходящего подхода предполагают использование «заглушек», заменяющих нереализованные модули нижних уровней, при отладке и тестировании модулей более высоких уровней. В самом простом случае «заглушка» может быть пустой, но она также может дополняться операторами печати. Такое дополнение позволяет «заглушке» печатать сообщения, отмечающие ее активацию, что обеспечивает трассировку программы. Трассировка программы дает полезную информацию для выработки стратегии оптимизации программы. Она позволяет проследить путь прохождения управляющего и информационного графов программы в зависимости от наборов начальных данных. Дополнив заглушки самыми тривиальными счетчиками, можно без дополнительных усилий уже на этапе создания программы получать информацию о частоте выполнения отдельных участков программы, т. е. выявлять наиболее «узкие», с точки зрения оптимизации, участки. Выделение таких участков на этапе создания программы до их кодирования может определять язык программирования для каждого конкретного участка программы. Действительно, наиболее «узкие», с точки зрения оптимизации, участки, можно запрограммировать на языках, для которых имеются эффективные трансляторы, в частности, можно использовать язык ассемблера. Для программ, к которым предъявляются не очень жесткие требования по эффективности, это может исключить дальнейшую специальную оптимизацию, что приводит к экономии как человеческих, так и машинных ресурсов.

Структурное программирование предполагает при записи алгоритмов программ и при их кодировании использование только нескольких стандартных структур. Выделяются три типа основных стандартных структур: следование, развилка и повторение. Исходя из общих соображений, можно предположить, что структурное программирование снижает качество получаемых программ. Действительно, с одной стороны, оно ограничивает множество конструкций, из которых составляются программы, а следовательно, и множество приемов и ухищрений, с помощью которых разработчик стремится повысить качество своей программы. С другой стороны, ограничение множества конструкций не дает возможности разработчику составлять (естественно несознательно) произвольные, в том числе и очень неоптимальные программы. Кроме того, получаемые за счет структурирования

наглядность, легкая читаемость и понимаемость программ позволяют повысить эффективность процесса оптимизации. Неоптимальность, получаемая (якобы) при структурном программировании, т. е. обусловленная составлением программы из фиксированных структур, является вполне контролируемой в отличие от неоптимальности, получаемой при традиционном подходе к программированию, которая никаким образом не попадает под контроль и определяется только фантазией и изощренностью разработчика программ. Вероятно, эти факторы и являются причиной отсутствия на практике сколь-нибудь значительных различий в эффективности между структурированными и неструктурированными программами. Можно сказать, что структурный подход помогает не создавать очень неэффективные программы [92].

Обусловленная структурным подходом к программированию дисциплина написания программ приводит к унификации методов оптимизации и, в частности, определяет применение специальных оптимизационных преобразований. Следует отметить следующие характерные особенности этих преобразований:

1. Структурное программирование упрощает и стандартизирует общую структуру управляющего графа программы, что приводит к упрощению алгоритмов анализа этого графа и, следовательно, к экономии машинных ресурсов. Для оптимизации такая экономия ресурсов, как правило, означает возможность более полного анализа управляющего графа, что приводит к повышению оптимизационного эффекта процедур, алгоритмы реализации которых основаны на этом анализе. Хорошей иллюстрацией сказанного являются алгоритмы анализа управляющего графа, ориентированные на глобальную экономию памяти, когда ограничения, связанные с машинными ресурсами, определяют возможности этой экономии.

2. Так как используемые структуры фиксированы и их число невелико, то число оптимизационных преобразований, применяемых к структурированным программам и ориентированных на преобразование используемых структур, значительно меньше числа оптимизационных преобразований, применяемых к неструктурированным программам и ориентированных на фрагменты, отражающие достаточно богатую фантазию программистов.

3. Применение только фиксированных структур позволяет полностью описать контекст, в котором они могут использоваться, что, в свою очередь, позволяет более полно разрабатывать алгоритмы оптимизационных преобразований и, следовательно, повышать их эффект.

Ограниченное множество преобразуемых структур, знание контекста и ограниченное множество оптимизационных преобразований позволяют:

1) более полно исследовать теоретические вопросы, связанные с последовательностью применяемых оптимизационных преобразований и, следовательно, выработать более эффективную стратегию оптимизации;

2) за счет стандартизации процесса оптимизации, как и всякой другой стандартизации, повысить производительность труда, т. е. повысить оптимизационный эффект; такая стандартизация особенно важна при управляемой оптимизации, когда у программиста имеются неограниченные возможности для повышения квалификации, т. е. умения находить нужную стратегию оптимизации для получения максимального эффекта.

Рассмотренные факторы, по мнению авторов, указывают на то, что структурное программирование предоставляет значительные возможности для выполнения эффективной оптимизации и, следовательно, позволяет в результате получать более качественные программы, чем программирование, основанное на традиционном подходе.

Пошаговая детализация. Процесс пошаговой детализации включает в себя разложение функции модуля на подфункции, которые в результате превращаются в шаги нужной программы. Этот процесс аналогичен нисходящему проектированию программы. Однако, если схема иерархии показывает функции и их подчинение, то пошаговая детализация применяется для декомпозиции функции каждого модуля в соответствии с внутренней логикой, необходимой для выполнения модулем этой функции. Пошаговая детализация — это итеративный процесс, в котором при каждом проходе по модулю рассматриваются все более подробные детали. Результатом пошаговой детализации является программа, написанная на псевдокоде.

Декомпозиция функций каждого модуля в процессе пошаговой детализации является, так же, как и при нисходящем проектировании, естественной факторизацией программы. Здесь факторизация, также в отличие от традиционной факторизации снизу вверх, применяемой при оптимизации программ, осуществляется сверху вниз. В соответствии с принципами пошаговой детализации сначала строится самый укрупненный граф модуля, вершинам которого соответствуют самые крупные подфункции, затем вершинам этого графа ставятся в соответствие графы, отражающие разбиение соответствующих подфункций на более мелкие подфункции, и т. д. Управляющие и информационные связи между вершинами графов различных уровней (здесь предполагается, что на каждом уровне есть два вида графов: управляющий и информационный) определяются реализацией на псевдокоде соответствующей подфункции, иначе говоря, сразу же в процессе пошаговой детализации, одновременно с выделением вершин графов, устанавливаются управляющие и информационные связи между этими вершинами. Преимущества такой факто-

ризации сверху вниз, осуществляемой в процессе пошаговой детализации, практически те же самые, какие были описаны при рассмотрении нисходящего проектирования программ, и поэтому здесь они подробно не обсуждаются. Следует только отметить следующий существенный фактор. Так как при выполнении пошаговой детализации используется псевдокод, то структура управляющих графов различных уровней подчиняется определенной дисциплине, а именно, дисциплине структурного программирования. Поэтому все, что было сказано про оптимизацию структурированных программ, можно отнести и к преобразованиям управляющих графов, получаемых в результате факторизации при выполнении пошаговой детализации, по сути дела преобразования структурированных программ являются и преобразованиями таких графов.

Один из принципов пошаговой детализации состоит в тщательном анализе используемых данных. С точки зрения оптимизации программ, следует отметить три аспекта этого принципа:

1. Тщательный анализ используемых данных позволяет вводить только самые необходимые данные, а это уже в некотором смысле является оптимизацией памяти.

2. Выбор данных ведется с учетом структуры алгоритма и проектируемой программы, более того, сами данные могут влиять на эту структуру. Поэтому структура алгоритма и программы и используемые данные максимально соответствуют друг другу, т. е. не существует никаких «лишних» данных по отношению к алгоритму и к программе. Это позволяет сокращать ресурсы, необходимые для оптимизации программ, так как отсутствуют затраты на оптимизацию «лишних» данных, которые, как правило, присутствуют в программах, созданных традиционными методами.

3. При анализе используемых данных в соответствии с принципами пошаговой детализации можно рассматривать некоторые оптимизационные оценки этих данных, например объем занимаемой ими памяти, ресурсы, необходимые для преобразования одних типов данных в другие и т. д. Такие оценки позволяют вводить данные и, следовательно, создавать программы с учетом оптимизационных критериев, т. е. способствовать повышению качества этих программ.

Другой принцип пошаговой детализации состоит в разбиении больших модулей на сегменты. Сегмент (в понимании структурного подхода) — это и логическая, и физическая часть модуля. Логически — это подфункция функции модуля. Физически сегмент ограничивается числом строк исходного текста, которое помещается на одной странице печатающего устройства. Каждый сегмент должен быть простой программой и возвращать управление своему старшему сегменту. Это позволяет читать и понимать любой сегмент без изучения сегментов более низкого уровня.

На первый взгляд, сегментирование, с точки зрения оптимизации программ, не способствует повышению качества программ по следующим причинам: выполнение команд, обеспечивающих вызов и возврат из сегмента, является функционально излишним; глобальная оптимизация программ, как правило, практически для всех оптимизационных процедур более эффективна на одном большом сегменте, чем на эквивалентных ему (в функциональном смысле) нескольких маленьких сегментах.

Однако следует учитывать, что сегмент является не только физической, но и логической частью модуля, поэтому между вызываемым и вызывающим сегментами существуют только самые необходимые управляющие и информационные связи. Это обусловливает то, что ресурсы, необходимые для выполнения команд, обеспечивающих вызов сегмента, являются минимальными и, в общем случае, меньшими, чем аналогичные ресурсы при традиционном подходе к программированию. Кроме того, сегментирование, определяемое структурным подходом к программированию, повышает наглядность получаемых программ.

Сегментирование отнюдь не означает разбиение программы на отдельные независимые транслируемые модули. Поэтому для некоторых языков программирования даже в случае разбиения модуля на сегменты возможно выполнение глобальной оптимизации в пределах всего модуля.

Здесь следует заметить, что псевдокод, на котором осуществляется пошаговая детализация, можно рассматривать как язык сверхвысокого уровня. На уровне этого языка возможно (после окончания пошаговой детализации) выполнение самостоятельной и достаточно разнообразной оптимизации, в процессе которой осуществляется преобразование алгоритма и структурной схемы программы (характер и возможности такой оптимизации были рассмотрены на с. 17).

Структурное программирование на языках высокого уровня возможно несколькими способами (здесь не рассматриваются языки программирования, специально ориентированные на структурное программирование, которые в настоящее время еще не получили достаточного распространения):

расширением синтаксиса языка путем добавления недостающих конструкций структурного программирования и использованием аппарата макросов, реализующих эти недостающие конструкции;

использованием предпроцессора, переводящего структурированные программы на язык высокого уровня;

моделированием конструкций структурного программирования средствами языка высокого уровня.

Все эти способы обладают, с точки зрения оптимизации программ, практически одними и теми же свойствами, основные из которых были рассмотрены на с. 128—130. Кроме того, струк-

турное программирование на конкретных языках обладает рядом дополнительных свойств, обусловленных некоторыми особенностями этих языков.

При использовании любого способа написания структурированные программы на языках высокого уровня, в соответствии с принципами структурного подхода к программированию, должны быть надежными и легко понимаемыми. Легкая понимаемость программ для различных языков достигается разными способами: наличием хороших комментариев, сегментированием, дополнением программы ее описанием на псевдокоде, разделением описательной и выполняемой части программы, соблюдением формата страниц, правильной нумерацией предложений программы, использованием отступов и строк продолжений и т. д. Легкая понимаемость программ разработчиками имеет существенное значение при оптимизации программ, так как способствует повышению ее оптимизационного эффекта.

При моделировании конструкций структурного программирования средствами языков высокого уровня возникает необходимость переоценить ряд типичных приемов программирования. Некоторые такие приемы противоречат принципам структурного программирования [92]. Кроме того, некоторые операторы языков высокого уровня следует использовать очень осторожно, либо вообще не использовать, так как их применение также противоречит принципам структурного программирования. Для ФОРТРАНа, например, такими операторами являются ENTRY, STOP, RETURN, GOTO, арифметический оператор IF, для ПЛ/1 — операторы ON, ENTRY, STOP, EXIT, RETURN.

Указанные факторы (моделирование конструкций структурного программирования средствами языка высокого уровня и, следовательно, употребление операторов языка высокого уровня в определенном контексте (в определенных структурах); отказ от использования некоторых типичных приемов программирования; отказ от использования некоторых операторов языков высокого уровня) обусловливают определенную дисциплину программирования на языках высокого уровня. Такая дисциплина программирования позволяет:

более точно определять набор применяемых оптимизационных процедур;

выработать более правильную стратегию применения оптимизационных процедур, включенных в этот набор;

реализовать новые оптимизационные процедуры, ориентированные на применение некоторых операторов, моделирующих конструкции структурного программирования, в заранее известном контексте (для неструктурированных программ реализация таких оптимизационных процедур может быть неэффективной).

Другими словами, эта дисциплина программирования позволяет ориентировать оптимизационные преобразования на конкретный язык программирования, т. е. «подогнать» оптимизационные преобразования не только под определенные структуры, но и под конкретный язык программирования, что, в свою очередь, позволяет еще больше повысить оптимизационный эффект последовательности выполняемых преобразований.

Кроме того, на основании принципов структурного подхода к программированию и некоторых особенностей языков высокого уровня вырабатывается определенный стиль программирования на этих языках, который увеличивает значимость некоторых оптимизационных процедур. Например, для ФОРТРАНа рекомендуется [92]: использовать каждый индикатор только с одной определенной целью; использовать имена для констант, определяя их значение с помощью объявления начальных данных; начальные значения переменным и индикаторам присваивать операторами, а не объявлениями начальных данных. Следование этим рекомендациям поднимает значимость таких оптимизационных процедур, как локальная и глобальная экономия памяти, подстановка, использование конструкции DATA.

Структурное программирование не рассматривает выбор трех основных структур как догму. Поэтому, если будут найдены некоторые другие структуры, отличные от традиционных (основных), применение которых не противоречит общим принципам структурного подхода к программированию, то такие структуры можно использовать так же, как и традиционные. Например, для языка ПЛ/1 в работе [92] рассматривается структура ПОИСК, суть которой состоит в просмотре таблицы и в поиске в этой таблице по ключу определенных элементов. Заметим, что такие структуры могут вводиться не только в языки программирования, но и в псевдокод. Использование этих новых структур, в общем случае, практически не влияет на выводы, сделанные в этом и в другом разделах, и все, что было сказано про основные структуры, может быть отнесено и к новым вводимым структурам.

Сквозной структурный контроль — это новое средство, направленное на облегчение управления проектом, которое, однако, по существу относится не к управленческому, а к техническому инструментарию, используемому разработчиками, непосредственно занятыми в проекте. Термин структурный отражает тот факт, что этот контроль является естественной составной частью процесса реализации проекта. Термин сквозной указывает на способ проверки — все контролируемые элементы мысленно выполняются шаг за шагом. На практике сквозной структурный контроль осуществляется путем организации контрольных сессий, проводимых в соответствии с определенными правилами. Одним из вопросов, который обсуждается на контрольных сессиях, является вопрос эффективности программ. Так как сквоз-

ной структурный контроль охватывает все этапы производства программ, начиная от проектирования и кончая изготовлением документации на готовый программный продукт, то в процессе осуществления этого контроля на всех этапах возможно принятие решений, направленных на повышение эффективности программ. Это значит, что в процессе сквозного структурного контроля возможно изменение как алгоритма, так и текста программы с целью повышения ее эффективности, т. е. осуществляется оптимизация на языках всех уровней, начиная от сверхвысокого уровня и кончая языками программирования. При этом контрольные сессии способствуют тому, что неоптимальные конструкции алгоритма и программы будут обнаружены на самых ранних стадиях развития проекта и поэтому их исправление потребует минимальных затрат ресурсов. Учет требований эффективности программы в процессе сквозного структурного контроля является одним из элементов постепенного перехода традиционного процесса оптимизации программ в процессе проектирования оптимальной программы. В этом смысле все аспекты сквозного структурного контроля могут быть практически без изменений перенесены на процесс проектирования оптимальной программы.

Особое значение сквозной структурный контроль, по всей видимости, может иметь при управляемой оптимизации, когда выработка оптимизации и принятие многих решений зависят в основном только от программиста и практически не ограничиваются никакими инструкциями. Обсуждение в соответствии с принципами структурного подхода к программированию на контрольной сессии принятых программистом решений, безусловно, будет способствовать повышению эффективности процесса оптимизации.

Следует также отметить следующий момент. Реализация оптимизации на языках различных уровней, реализация управляемой оптимизации, внесение понятий структурного подхода к программированию в оптимизацию программ значительно повышают роль программиста в осуществлении процесса оптимизации. И здесь многое зависит от квалификации этого программиста. Контрольные сессии, где встречаются и участвуют в обсуждении программисты различной квалификации, безусловно, будут способствовать профессиональному росту этих программистов и тем самым способствовать повышению эффективности процесса оптимизации.

Тестирование — это процесс, посредством которого проверяется правильность программы. Цель тестирования — показать, что программа правильна и действительно работает в соответствии с проектными спецификациями на заданных наборах начальных данных. Тестируют, как правило, спецификации, конкретные модули и межмодульные связи. Общим для всех этих случаев является то, что в процессе тестирования осуществляется

либо полное, либо частичное (частичное в том смысле, что некоторые модули заменены «заглушками») выполнение программы, т. е. осуществляется прохождение управляющего графа программы по некоторым его путям. Этап тестирования можно использовать для целей оптимизации программ. Действительно, в процессе прохождения по путям управляющего графа программы можно при незначительной модификации отдельных модулей получить следующую информацию, необходимую для выбора более эффективной стратегии оптимизации:

1. Полный либо частичный «частотный профиль» программы и время ее выполнения, что позволяет выделить наиболее часто выполняемые участки (наиболее важные для оптимизации). Причем в процессе тестирования «частотный профиль» может быть построен как для всей совокупности программ, когда элементарной единицей, для которой определяется частота ее выполнения, является модуль, так и для отдельного модуля, когда элементарной единицей, для которой определяется частота ее выполнения, является один или несколько операторов.

2. Предварительные результаты выполнения программы. На основании таких результатов можно установить, в частности, диапазон изменения промежуточных и выходных данных программы. Такая информация определяет целесообразность выполнения некоторых оптимизационных процедур и, следовательно, на ее основе можно выбрать более эффективную стратегию оптимизации.

Некоторые другие аспекты, с точки зрения оптимизации программ, сочетания нисходящей разработки тестирования в рамках структурного подхода к программированию рассмотрены на с. 127—128. Здесь следует отметить, что «частотный профиль» программы, время ее выполнения и предварительные результаты можно получить только в процессе тестирования без ориентации на процесс оптимизации (не рассматривается модификация модулей, необходимая для получения этой информации, которую можно легко автоматизировать). Другими словами, при структурном подходе к программированию для получения рассматриваемой информации, которую необходимо получить для эффективного выполнения процесса оптимизации, практически не требуется никаких дополнительных ресурсов.

Общие аспекты. Структурный подход к программированию, безусловно, способствует повышению эффективности процесса оптимизации программ. Признано, что программы, написанные на языках высокого уровня, таких, как АЛГОЛ, ПЛ/1 и КОБОЛ, потенциально оказываются более эффективными при использовании принципов структурного программирования [53]. Эффективность программы, написанной на языке высокого уровня, зависит от качества объектного кода, формируемого компилятором с такого языка (если, конечно, не принимать во внимание

эффективность или неэффективность программы, заложенные в ней как внутренние свойства, что часто оказывается значительно важнее рассматриваемых деталей кодирования). Создатели компиляторов начали обращать внимание на то, что глобальная оптимизация (т. е. совместная оптимизация нескольких операторных выражений высокого уровня описания или оптимизация всей программы в целом) значительно проще в случае структурированных программ, так как управление такими программами может анализироваться автоматическими средствами. Достигаемый при оптимизации выигрыш значительно превышает все виды неэффективностей, внутренне присущих принципам структурного программирования и если этого не удастся достичь средствами современных компиляторов, то, безусловно, можно будет добиться компиляторами нескольких будущих лет [53].

Развитие методов оптимизации и новые теоретические идеи, появившиеся в этой области, свидетельствуют о том, что процесс получения качественной программы посредством оптимизирующего транслятора преобразуется в процесс проектирования качественной программы. Об этом, в частности, свидетельствуют оптимизация программ на языках сверхвысокого и последующих уровней, управляемая оптимизация, все более глубокое понимание того, что выбор стратегии получения оптимальной программы должен быть возложен на человека и как следствие развитие методов автоматизированной оптимизации и изучение повторных и тупиковых свойств оптимизационных процедур. Действительно, такие же или аналогичные понятия являются неотъемлемыми компонентами процесса проектирования программ. Более того, принципы многоуровневого проектирования программ в явном или неявном виде признаны и в оптимизации программ [32]. Такое понимание процесса оптимизации программ (проектирование оптимальной программы), а также совпадение различных этапов процесса оптимизации программ и структурного подхода к программированию и совпадение областей их применения определяют то общее, что позволяет судить о близости и взаимном влиянии этих двух направлений программирования. Кроме того, эти же факторы позволяют перенести методы структурного подхода к программированию, повышающие эффективность процесса проектирования программ, на процесс оптимизации программ для повышения его эффективности. Некоторые действия процесса создания программ в соответствии со структурным подходом к программированию и некоторые действия процесса оптимизации программ совпадают или почти совпадают. Это значит, что ресурсы, затрачиваемые при структурном подходе к программированию для достижения хорошей надежности и требуемого качества программ, меньше ресурсов, затрачиваемых для достижения той же надежности и качества при традиционном подходе к программированию. Кроме того, косвенное

сокращение ресурсов, затрачиваемых на оптимизацию программ, возникает вследствие повышения надежности программ, созданных в соответствии со структурным подходом к программированию. Действительно, повышение надежности уменьшает вероятность того, что будет найдена ошибка, т. е. программа будет модифицироваться, а следовательно, будет повторена оптимизация. Такая экономия ресурсов, в свою очередь, позволяет повысить оптимизационный эффект выполняемых преобразований, так как в некоторых случаях этот эффект находится в прямой зависимости от выделяемых ресурсов.

Повышению эффективности процесса оптимизации способствует также то, что структурный подход предоставляет реальные возможности осуществления оптимизации программ на языках сверхвысокого уровня. В частности, такими языками могут быть язык, используемый на этапе нисходящего проектирования программы, и ПСЕВДОКОД. В таком случае процесс оптимизации программ распределен по следующим уровням: язык, используемый при нисходящем проектировании программ; псевдокод; язык высокого уровня; промежуточный язык; язык ассемблера.

Осуществление оптимизации на языках сверхвысокого уровня предоставляет дополнительные возможности для повышения эффективности процесса оптимизации. Говоря об оптимизации на уровне языка, используемого при нисходящем проектировании программ, следует отметить следующий момент, подчеркивающий тесную связь структурного подхода к программированию и оптимизации программ: на этапе нисходящего проектирования программ строится граф программы, однако на этом этапе выделяются только вершины этого графа, а управляющие и информационные связи между вершинами устанавливаются только на этапе записи программы на псевдокоде или этапе ее кодирования. После построения графа над ним выполняется ряд оптимизационных преобразований. В результате выполнения этих преобразований может быть найдено, что программа, с точки зрения оптимизации программ, спроектирована неэффективно и ее необходимо перепроектировать; причем программу необходимо перепроектировать даже несмотря на то, что были выполнены этапы нисходящего планирования и реализации, пошаговой детализации и т. д. Однако это не противоречит принципам структурного подхода к программированию, и более того, такое перепроектирование, т. е. возвращение на более высокий уровень языка, если будет найдено более удачное решение, является одним из принципов этого подхода, а приведенный пример показывает, как процесс оптимизации программ смыкается и помогает более эффективно реализовать принципы этого подхода.

Превращение процесса оптимизации программы в процесс проектирования оптимальной программы позволяет учитывать

на различных этапах создания программы и записи ее на языках различных уровней требования эффективности и соединить эти требования, возникающие на различных этапах, в единое целое. В частности, при переводе программы с языка высокого уровня на язык более низкого уровня, например с псевдокода на ПЛ/1, а затем на язык ассемблера, можно дополнить программу информацией (которая будет сохранена в процессе ее перевода вплоть до языка ассемблера), используемой для оптимизации программ на уровне языка ассемблера. Такая информация может включать в себя: указание о том, какие формальные параметры являются входными, какие выходными; перечисление подпрограмм, в которых используется та или другая глобальная переменная; описание управляющих связей между подпрограммами и т. д.

Структурное программирование, являющееся одним из компонентов структурного подхода к программированию, предполагает составление программы из набора основных структур трех типов, который при необходимости можно расширить, например, введя структуру ЭФ-ВЫБОР (эффективный выбор), являющуюся обобщением структуры ЕСЛИ—ТО—ИНАЧЕ и отличающуюся от структуры ВЫБОР тем, что в ней запрещено одновременное использование сразу на всех альтернативах семантически одинаковых операторов присваивания. Если необходимо такое использование этих операторов, то один из них помещается перед структурой ЭФ-ВЫБОР.

Структурный подход к программированию меняет значимость и частоту применения некоторых оптимизационных процедур. При нем не рекомендуется использование общей памяти, так как это затрудняет понимаемость программы. Однако, из-за требований эффективности часто возникает потребность в использовании такой памяти. Поэтому при следовании структурному подходу к программированию повышается значимость оптимизационных процедур, выполняющих оптимизационные преобразования, связанные с общей памятью модулей. Другими словами, при структурном подходе к программированию, вне зависимости от фантазии и изощренности программиста, можно выделить некоторое конкретное подмножество применяемых оптимизационных процедур. А это также способствует повышению эффективности процесса оптимизации, так как позволяет сконцентрировать внимание и усилия разработчиков на решении теоретических и практических вопросов оптимизации, связанных только с этим конкретным подмножеством процедур.

Следует отметить влияние структурного подхода к программированию на такое направление оптимизации программ, как управляемая оптимизация. Безусловным является то, что рассматриваемый подход создает дополнительные возможности для повышения эффективности управляемой оптимизации. В этом

смысле следует отметить два фактора: легкую читаемость и понимаемость создаваемых программ и наличие дополнительной информации о программе, получаемой в процессе ее создания в соответствии только с принципами структурного подхода к программированию и безотносительно к процессу оптимизации. Легкая читаемость и понимаемость программ и наличие такой дополнительной информации позволяют разработчику точнее оценить особенности программы и, следовательно, выбрать более правильную стратегию ее дальнейшей оптимизации, что, естественно, приводит к повышению оптимизационного эффекта. Кроме того, легкая читаемость и понимаемость программ помогают программисту самому в процессе создания программы обнаружить некоторые неоптимальные места и исправить их, а это делает излишним выполнение некоторых оптимизационных процедур (таких процедур, необходимость выполнения которых обусловлена сложностью и запутанностью структуры программы и трудностью ее понимания).

Таким образом, структурированные программы предоставляют больше возможностей для выполнения эффективной оптимизации, чем неструктурированные, и, следовательно, позволяют получать более качественные программы. Это, в частности, связано с тем, что во многих своих аспектах структурный подход к программированию способствует повышению эффективности оптимизации по тем же причинам, по которым этот подход способствует повышению надежности программ. Способы, с помощью которых повышается эффективность оптимизации, отличны от традиционных (точно так же, как и сам структурный подход отличен от традиционного), и не всегда можно явно выделить «механизм», обеспечивающий повышение качества программ. Однако совокупность этих методов, объединенных идеями структурного подхода к программированию, обеспечивает повышение эффективности оптимизации. Имеющееся, на первый взгляд, отрицательное влияние структурного подхода к программированию на оптимизацию программ связано, в основном, с принципом модульности, который является одной из основ рассматриваемого подхода. Но принцип модульности присутствует в любом подходе к написанию больших программ и, более того, при написании больших программ в настоящее время не существует альтернативы этому принципу. Структурный подход к программированию позволяет наиболее эффективным способом реализовать принцип модульности, в частности, этот подход обеспечивает максимальную независимость получаемых модулей. Другими словами, отрицательное влияние на оптимизацию программ принципа модульности в структурном подходе к программированию является минимальным по сравнению с другими подходами к программированию.

4. СТРУКТУРА СИСТЕМ ОПТИМИЗАЦИИ ПРОГРАММ

Структура систем оптимизации программ и их внутренняя организация определяются следующими двумя основными факторами: реализацией оптимизационных преобразований и назначением системы оптимизации программ.

Способы реализации оптимизационных преобразований. В настоящее время можно выделить два способа реализации оптимизационных преобразований.

Первый способ состоит в реализации алгоритмов, осуществляющих оптимизационные преобразования прикладных программ на языках различных уровней (входного, промежуточного, ассемблера). Эти алгоритмы структурируются в виде отдельных оптимизационных блоков (называемых оптимизационными процедурами) и независимы от других процессов преобразования программ (трансляции, отладки и т. д.). Оптимизационные процедуры осуществляют анализ управляющих и информационных связей прикладных программ. Затем на основе проведенного анализа они преобразуют программы в более качественные и, вместе с тем, осуществляющие те же функции, причем уровень языка, на котором программы составлены, не меняется. После этого осуществляется трансляция программ на язык более низкого уровня. Характерной чертой оптимизационных процедур является то, что они не затрагивают собственно процесса трансляции и их можно (либо в принципе существует такая возможность) отключать. Такие процедуры выполняют, как правило, устранение различного типа избыточных вычислений. Примером системы, в которой осуществляется этот подход к реализации оптимизационных преобразований, является БЕТА — многоязыковая система программирования [43].

Второй способ реализации оптимизационных преобразований состоит в разработке и реализации эффективных алгоритмов трансляции, позволяющих повышать качество генерируемых объектных кодов. Такие алгоритмы, как правило, обеспечивают эффективное распределение памяти и регистров, уменьшают число обменов между сумматором и различными видами памяти, выбирают эффективные машинные коды и т. д. В этом случае процесс повышения качества программ осуществляется в процессе собственно трансляции программы с одного языка на другой, т. е. алгоритмы оптимизационых преобразований представляют собой часть процесса трансляции, и их нельзя отключить во время работы транслятора. Кроме того, такие алгоритмы оптимизационных преобразований являются, как правило, частью семантики входного языка и задаются либо в формальном описании его семантики, либо при разработке семантических подпрограмм. При реализации этих алгоритмов нельзя говорить об уровне языка, на котором осуществляется оптимизация, так как оптимизационные

преобразования как бы «размазаны» по всему процессу трансляции. В качестве примера можно привести АЛЬФА-систему автоматизации программирования [2], где такие алгоритмы выполняются и в процессе декомпозиции исходной программы, и в процессе генерации, и поэтому неотделимы от процесса трансляции. Характерной чертой второго способа реализации оптимизационных процедур является широкое применение «смешанной» стратегии, когда один самый общий алгоритм трансляции дополняется серией менее общих, рассчитанных на конкретный контекст алгоритмов. Применение таких менее общих алгоритмов позволяет для этого контекста генерировать эффективные объектные коды.

Следует отметить, что в настоящее время наиболее распространенным является первый способ реализации оптимизационных преобразований. Это обусловлено двумя причинами:

1. Отделение алгоритмов оптимизации от алгоритмов трансляции позволяет разработчикам при их реализации концентрировать свои усилия отдельно на каждом из этих множеств алгоритмов. Это, в свою очередь, позволяет, в частности, реализовать более сложные алгоритмы оптимизационных преобразований, т. е. расширить набор реализуемых процедур и повысить их оптимизационный эффект.

2. В связи с алгоритмической неразрешимостью задачи оптимизации процесс генерации оптимальной программы все больше приобретает черты автоматизированного конструирования оптимальной программы. Автоматизированное конструирование предполагает, что общую стратегию оптимизации вырабатывает программист, а ЭВМ выполняет рутинную работу, причем общение программист — ЭВМ осуществляется в процессе диалога. При этом в процессе оптимизации возможны попадание в тупиковую ситуацию и вследствие этого возврат на некоторый промежуточный этап, причем возврат может осуществляться с языка низкого уровня на язык более высокого уровня. Все это выполняется наиболее эффективно при первом способе реализации оптимизационных преобразований; второй способ более характерен для оптимизирующих трансляторов, созданных в шестидесятых годах, хотя в некоторых случаях его используют и при создании современных систем.

Направления создания систем оптимизации программ. В настоящее время универсальные языки высокого уровня обнаруживают полезную и приятную для программистов стабильность. Действительно, множество этих языков устоялось, уточнены формулировки и устранены неточности понимания, разработаны специальные приемы написания программ на этих языках, созданы богатые библиотеки подпрограмм. Языки приняты и широко используются программистами различных направлений и квалификации.

Одновременно, в связи с быстрым расширением сферы приме-

нения ЭВМ в самых различных областях человеческой деятельности, на передний план выдвигается задача создания в широких масштабах проблемно-ориентированного программного обеспечения. Универсальное программное обеспечение, предназначенное в первую очередь для программирования и решения потока независимых задач, в ряде случаев не удовлетворяет разнообразным, подчас противоречивым, требованиям пользователей ЭВМ. Удельный вес задач, требующих проблемно-ориентированных диалектов или специальных языков программирования, постоянно увеличивается. Это обусловлено, как правило, экономическими и/или техническими причинами, связанными с направленностью используемых языков на решение только одного узкого класса задач (или, может быть, отдельной задачи) и ограниченностью времени существования используемых языков временем решения этого класса задач (или отдельной задачи). Разработка и реализация новых языков за прошедшие годы интенсивно продолжалась [113] и продолжается в настоящее время.

Рассмотренные тенденции обусловливают два ярко выраженных направления создания систем оптимизации программ. Одно направление ориентировано на создание систем оптимизации программ для широко распространенных и установившихся языков программирования, таких как ПЛ/1, АЛГОЛ-68, АЛГОЛ-60, Паскаль, ФОРТРАН. Другое направление ориентировано на создание систем оптимизации программ для специальных языков программирования, множество которых постоянно меняется и расширяется.

Выделение рассмотренных двух направлений создания систем оптимизации программ согласуется с общей тенденцией развития системного программирования, основную задачу которого в настоящее время можно сформулировать как задачу создания промышленной технологии изготовления [27] процедурного (универсального) и проблемно-ориентированного программного обеспечения.

Архитектура систем оптимизации программ. Для каждого из указанных направлений создания систем оптимизации программ характерна своя архитектура.

Основной характеристикой, определяющей структуру систем оптимизации программ первого направления, является его замкнутость, т. е. наличие одного или нескольких фиксированных входных языков. Небольшое число входных языков, с одной стороны, и их высокий уровень, с другой, обусловливают ступенчатую схему трансляции, которая и является основой систем оптимизации программ первого направления, причем оптимизационные преобразования выполняются, как правило, на уровне промежуточных языков. Поэтому промежуточный язык выбирают с учетом не только особенностей входных языков и кодов объектной ЭВМ (здесь под объектной понимается ЭВМ, на кото-

рой выполняется сгенерированная программа), но и с учетом требований реализации алгоритмов оптимизационных преобразований. Для систем первого направления одна и та же ЭВМ, как правило, является одновременно инструментальной (т. е. ЭВМ, на которой реализована система) и объектной. Например, на рис. 22 показана структура системы оптимизации первого направления с несколькими (K) входными языками высокого уровня и с генерацией кода на несколько (N) объектных ЭВМ.

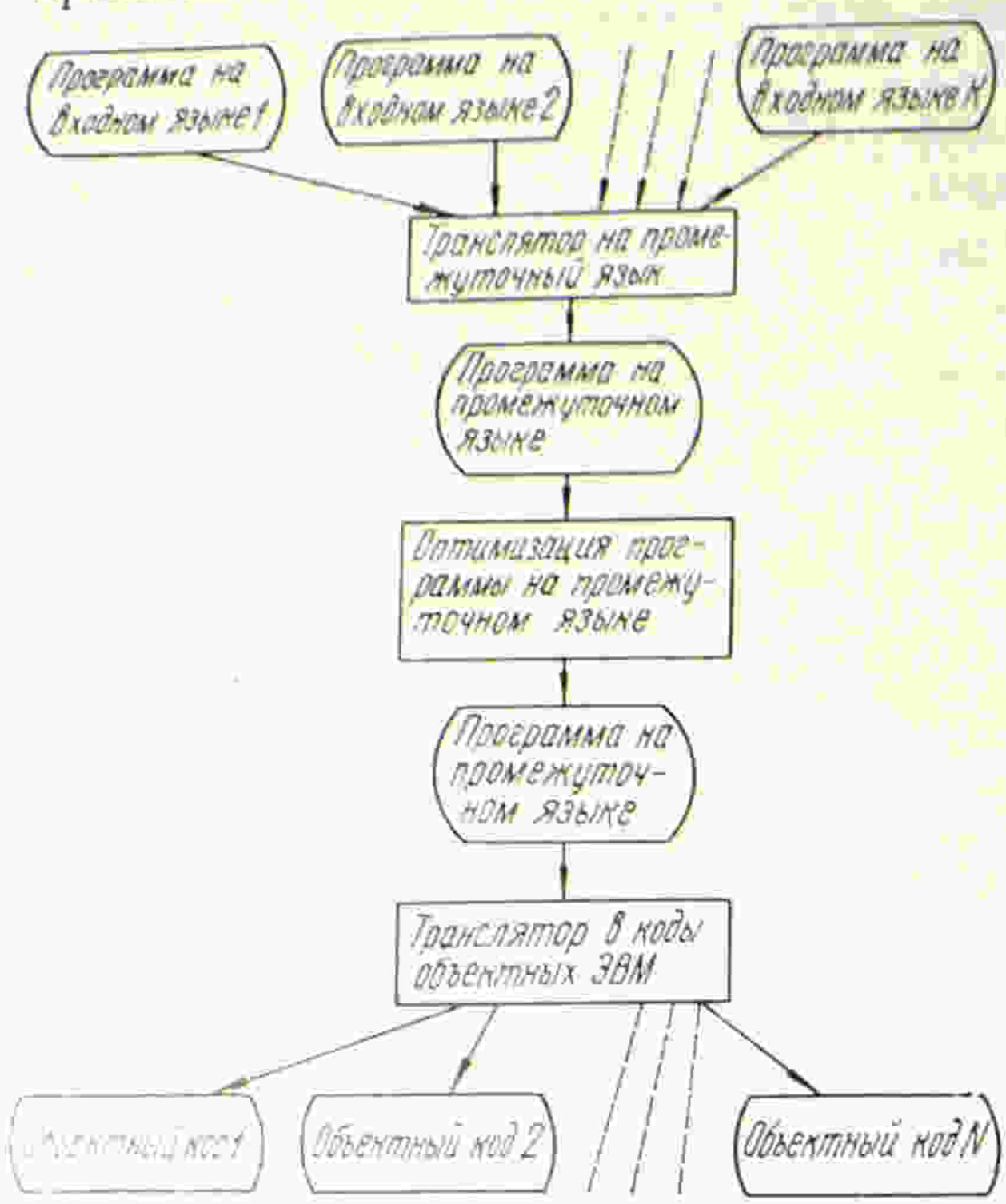


Рис. 22

Основной характеристикой, определяющей структуру систем оптимизации программ второго направления, является его открытость, т. е. тот факт, что множество входных языков таких систем не является стабильным, постоянно меняется и расширяется. Формализовать детали всех этих языков не представляется возможным. Практически представляется возможным сформулировать только самые общие характеристики некоторого подмножества специализированных языков, ориентированных на решение какого-то одного класса задач. Такие характеристики являются, как правило, описанием некоторой базы этих языков, например вычислительной части. Поэтому принятый подход к созданию систем оптимизации программ второго направления сводится к созданию универсальных оптимизирующих систем (оптимизаторов) с параметрической настройкой на входной язык.

Методика создания параметрически настраиваемых систем программирования в настоящее время хорошо разработана [111, 50]. Создан ряд синтаксически управляемых трансляторов: с языка РЕФАЛ, RTK; транслирующая система СТАРТ и другие, которые могут быть использованы для настройки систем оптимизации программ на входной язык. Более трудной и новой задачей является разработка принципов построения универсальных оптимизаторов, которые в результате параметрической настройки повышают качество программ. Для создания таких оптимизаторов необходимо создать модели программ, составленных на языках, являющихся входными, и разработать методы гло-

бальной оптимизации таких моделей. Разработка методов глобальной оптимизации для систем оптимизации программ второго направления имеет принципиальное значение, так как возможности локальной оптимизации в универсальных оптимизаторах уменьшаются. Действительно, локальная оптимизация, как правило, основана на свойствах конкретных языков программирования, которые при создании универсальных оптимизаторов, естественно, учитываться не могут.

Основой для создания универсального оптимизатора является метод построения областей значений переменных, разработанный для операторных схем [3, 5, 10, 12]. Операторные схемы — это модели программ, они определяются только на семантическом уровне, что отличает их от промежуточных языков, используемых в системах оптимизации программ первого направления. Например, синтаксические представления программ в операторных схемах, получаемых при различных входных языках, отличаются друг от друга. Определение операторных схем только на семантическом уровне облегчает процесс параметрической настройки универсальных оптимизаторов, так как практически отсутствует фаза перевода исходных программ в единое промежуточное представление и обратно. Метод построения областей значений переменных позволяет выделять в операторных схемах множества операторов, в пределах которых переменные сохраняют свои значения. Параметры, характеризующие области значений переменных, содержат информацию об управляющих и информационном графах множества операторных схем. Анализ областей значений переменных позволяет выявить взаимозависимости между значениями переменных для участков программ достаточно сложной структуры и поэтому позволяет выполнять глобальную оптимизацию программ. Управляющие и информационные связи между выделенными множествами операторов, являющимися областями значений переменных, служат базой для всех последующих преобразований, выполняющих глобальную оптимизацию программ, в частности, таких преобразований, как устранение невыполняемых и идентичных операторов и неиспользованных определений, оптимизация выражений, подстановка, чистка циклов и фрагментов, замена одних операций другими, более эффективными. Кроме того, анализ областей значений переменных позволяет выделять в программах различные структуры, такие, как зоны, интервалы, гамаки, фрагменты, и осуществлять факторизацию программы.

Например, на рис. 23 приведена структура системы оптимизации программ второго направления, выполняющей оптимизацию на уровне входного языка. Здесь параметрическая настройка на входной язык, осуществляемая на основе информации о его синтаксисе, необходима для функционирования двух подсистем, осуществляющих преобразование: программ, написанных на входном языке, в операторные схемы и операторных схем в программы, составленные на входном языке. Штриховой линией выделено ядро рассматриваемой

системы, состоящее из двух подсистем: транслятора, осуществляющего преобразование программ, написанных на входном языке, в операторные схемы, и универсального оптимизатора.

Семантический характер операторных схем и их общность позволяют их использовать в качестве моделей широкого класса языков различных уровней. Поэтому можно говорить об уровне операторных схем и о трансляции операторных схем одного уровня в операторные схемы другого уровня. Так, операторные схемы высокого уровня могут являться моделью входного языка, более низкого уровня — моделью промежуточного языка и еще более низкого — моделью языка ассемблера объектной ЭВМ. Это позволяет осуществлять параметрическую настройку ядра системы оптимизации программ на различные входные языки, которыми могут быть и операторные схемы, и на основе такой

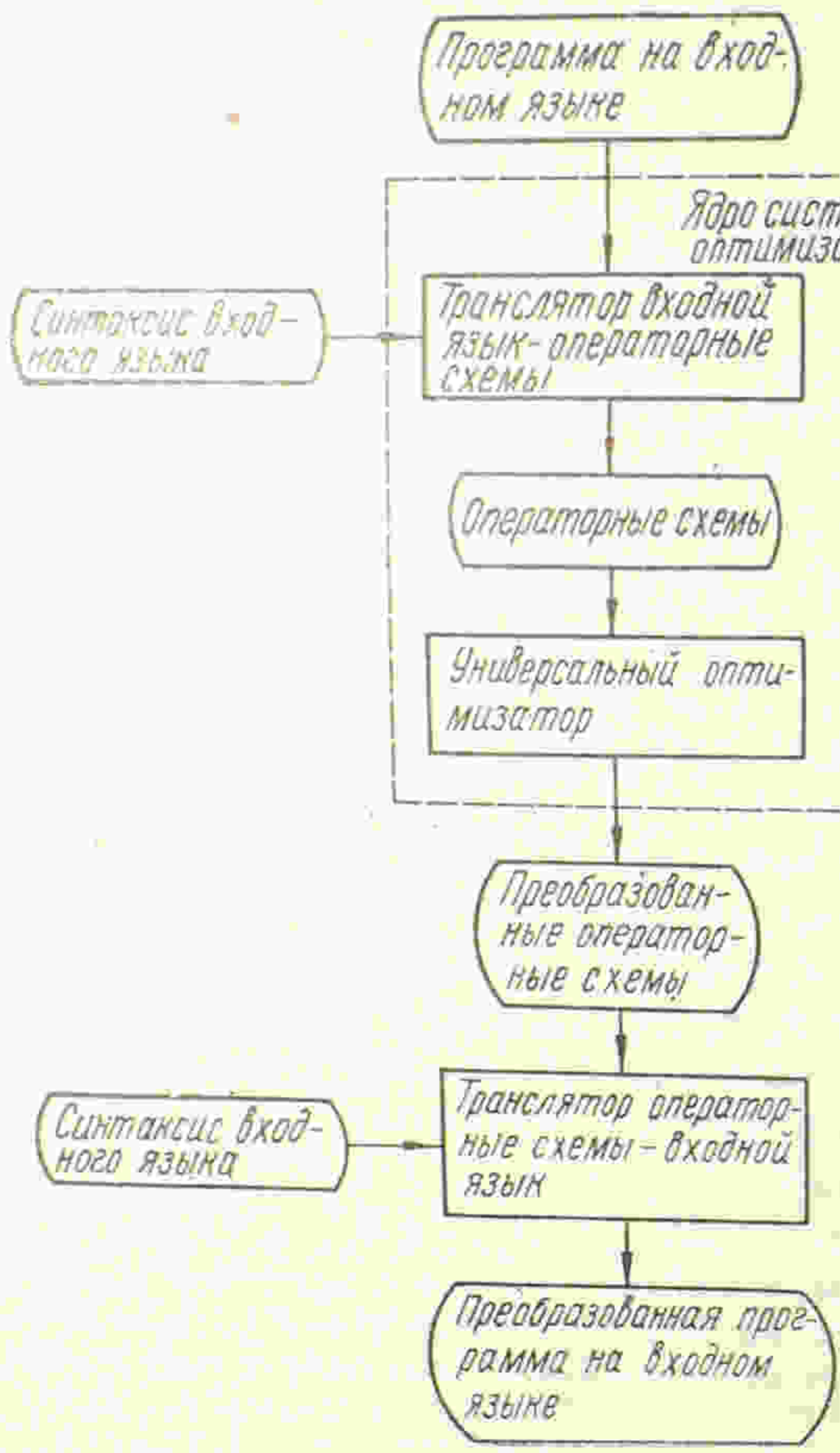


Рис. 23

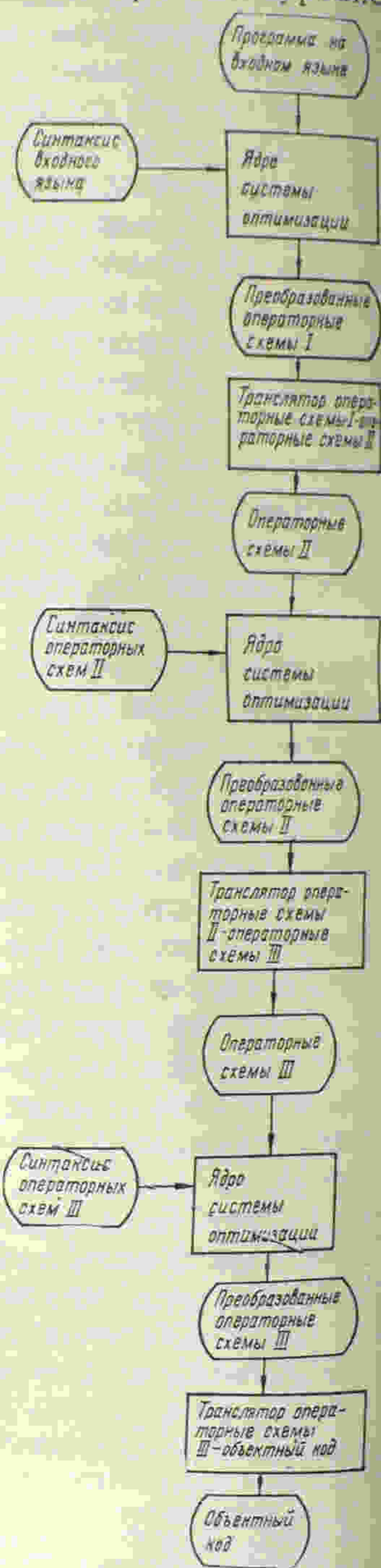


Рис. 24

настройки конструировать более сложные системы оптимизации программ второго направления, выполняющие как собственно оптимизацию, так и трансляцию программ с входного языка в коды объектной ЭВМ.

На рис. 24 показана такая трехступенчатая система оптимизации программ, осуществляющая одновременно с оптимизацией трансляцию программ, написанных на входном языке высокого уровня, в коды объектной ЭВМ. Здесь *операторные схемы I* являются моделью входного языка высокого уровня; *операторные схемы II* — моделью некоторого промежуточного языка; *операторные схемы III* — моделью объектных кодов. Заметим, что промежуточный

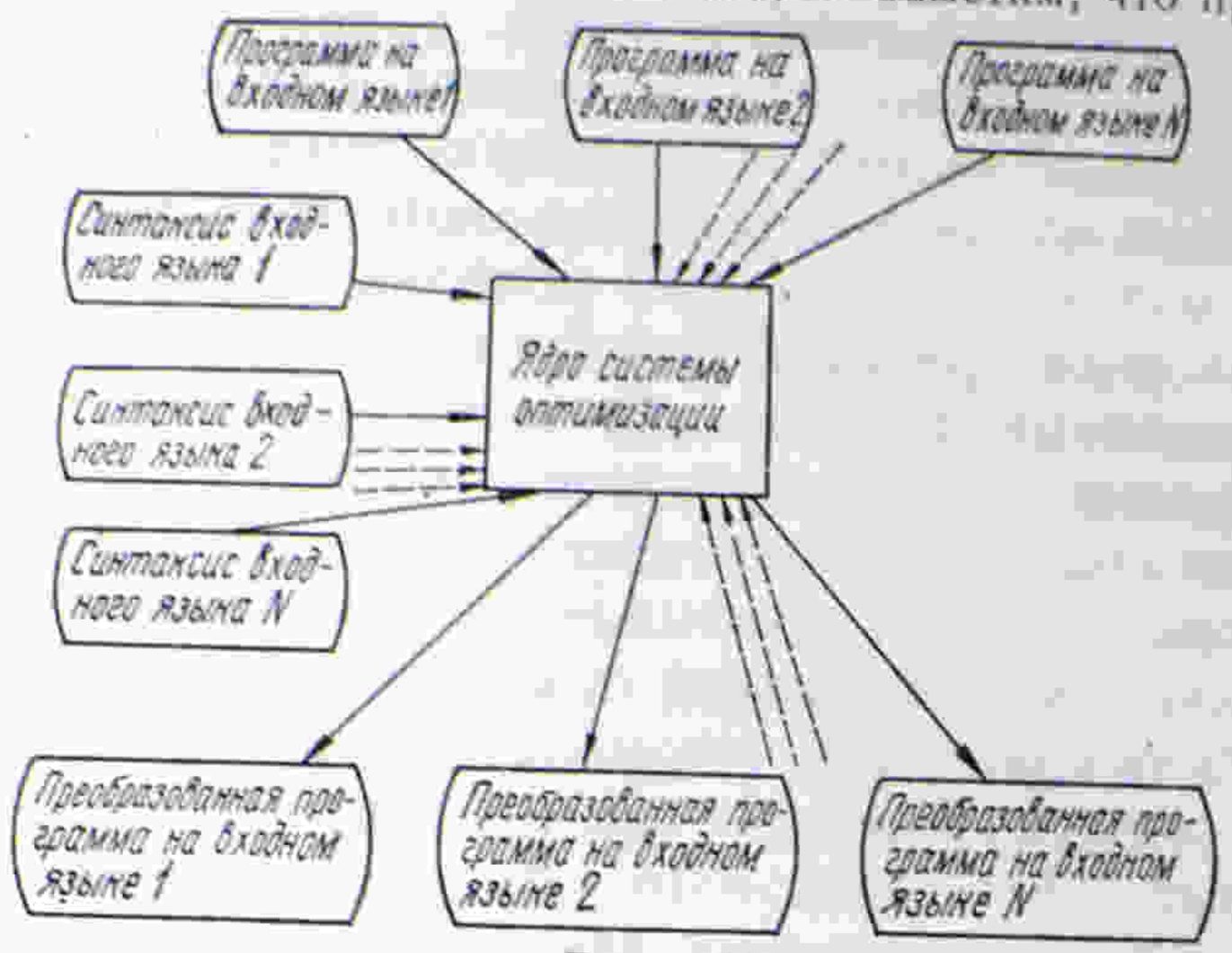


Рис. 25

язык как таковой отсутствует. В системе имеется только модель промежуточного языка, определяемая синтаксисом входного языка и синтаксисом объектных кодов, т. е. эти модели для различных входных языков и разных ЭВМ различны. На всех ступенях процесса оптимизации описанной системы используется одно и то же *ядро системы оптимизации*, настраиваемое на синтаксис различных языков. Так как на вход второй и третьей ступени рассматриваемой системы поступают *операторные схемы*, то *транслятор*, входящий в состав *ядра системы оптимизации* программ, осуществляет только настройку ядра на эти операторные схемы без выполнения их последующей трансляции.

Параметрическая настройка ядра системы на входной язык позволяет также создавать системы оптимизации программ с параллельной оптимизацией на нескольких входных языках одного уровня.

На рис. 25 показана структура такой системы, осуществляющей оптимизацию программ, написанных на одном из N языков, на синтаксис которого настроено *ядро системы оптимизации*. Отличие рассматриваемой системы от аналогичных систем оптимизации программ первого направления, осуществляющих оптимизацию программ, написанных на разных входных языках (см., например, рис. 22), заключается в том, что здесь входные языки не фиксированы и могут быть заменены другими настройкой на них ядра системы.

Процесс оптимизации, выполняемый универсальным оптимизатором, состоит, как правило, из двух этапов выполнения: глобального анализа управляющего и информационного графов

программы и сбора информации, получаемой в результате этого анализа для последующей оптимизации; оптимизационных процедур, т. е. осуществления собственно оптимизации. Выполнение на первом этапе глобального анализа управляющего и информационного графов осуществляется методом построения областей значений переменных. Набор оптимизационных процедур, выполняемых на втором этапе, может быть различным для различных моделей программ, но, как правило, он состоит из общепринятых процедур. Для операторных схем набор таких общепринятых процедур был указан выше.

Ядро системы оптимизации включает, как правило, процедуры, выполняющие только глобальную оптимизацию. Поэтому, для достижения максимально возможного качества программ ядро системы оптимизации можно дополнить блоком, выполняющим локальные (в пределах линейного участка или одного оператора) оптимизационные преобразования, ориентированные на конкретный язык программирования или на конкретные операторные схемы (здесь не рассматривается структура таких блоков и выполняемые ими оптимизационные преобразования, так как они определяются спецификой конкретного языка программирования или операторных схем).

Система оптимизации программ второго направления, показанная на рис. 24, помимо ядра системы оптимизации содержит три транслятора с различных языков. Эти трансляторы можно заменить одним из указанных выше синтаксически-управляемых трансляторов. Поэтому, используя два функциональных блока — ядро системы оптимизации и синтаксически-управляемый транслятор (или, быть может, только универсальный оптимизатор и синтаксически-управляемый транслятор), параметрически настраиваемые на входные языки, можно создавать достаточно универсальные системы оптимизации программ второго направления.

Оптимизация программ на уровне входных языков. Под входными языками понимаются языки высокого уровня, в частности, процедурно-ориентированные языки. Оптимизации программ на уровне входных языков уделяется особое внимание, в связи с тем, что, с одной стороны, системы оптимизации программ совершенствуются, в том числе и входные языки, а, с другой, эта область исследований является менее всего изученной. Исторически сложилось так, что первоначально изучение оптимизации программ осуществлялось на уровне объектных кодов, затем промежуточного языка и лишь в последние годы — на уровне языков высокого уровня. О необходимости выделения входных языков как уровня оптимизации свидетельствуют следующие факторы. Если промежуточное представление действительно является промежуточным, т. е. если оно не совпадает как с входным, так и выходным языками, то существует опасность, что

после перевода программы в промежуточное представление могут нарушиться или потеряться ее эффективно используемые свойства, а это является причиной порождения неэффективного кода и усложняет оптимизацию. Рассмотрение программы на более низком уровне и попытка точного определения тех свойств, которые явно содержались во входной программе, может окончиться безуспешно. Например, потеря информации о блочной структуре программы или оператора может привести к увеличению числа информационных связей в программе, обнаруживаемых при анализе; операторы цикла, организованные с помощью специальных средств и операторов условного перехода, могут транслироваться в одну и ту же программу на промежуточном языке, что затем усложняет анализ управляющего и информационного графов этой программы.

Современное программирование характеризуется появлением языков сверхвысокого уровня (см. гл. 1.2). В частности, в качестве такого языка можно рассматривать псевдокод [38]. Появление языков сверхвысокого уровня приводит к тому, что процедурно-ориентированные языки становятся, в свою очередь, промежуточными. Поэтому даже при следовании традиционному подходу к оптимизации программ, когда все преобразования выполняются на уровне промежуточных языков, значимость развития методов оптимизации на уровне процедурно-ориентированных языков непрерывно повышается.

Одним из наиболее перспективных направлений развития архитектуры современных ЭВМ является создание процессоров, в которых программные уровни внутренних языков машин приближены к алгоритмическим языкам программирования, а также развитие остальных их уровней для обеспечения совершенной организации системы ступенчатого микропрограммного управления [29]. Основными достоинствами такого развития внутренних языков являются облегчение и ускорение всего процесса подготовки задач, а также существенное повышение производительности процесса их решения (в особенности в режиме диалога) при использовании алгометрических языков, причем эти достоинства имеют силу не только для малых, но, по-видимому, еще в большей степени для больших машин коллективного пользования. На малых машинах серий МИР и «Наири» эти достоинства вполне проверены как качественно, так и количественно. В отношении же больших машин проведенные теоретические, а также экспериментальные исследования и практические разработки у нас в стране и за рубежом, позволяют ожидать весьма хороших результатов на этом пути развития. В качестве примера можно привести автокод многомашинного вычислительного комплекса «ЭЛЬБРУС» [15, 17]. Дальнейшим развитием данного направления следует считать приближение программных уровней внутренних языков не только к узкопроблемно-ориентированным, но и к

более универсальным языкам программирования, представителями которых являются ПЛ/1 и АПЛ (в его полном объеме). Одной из самых известных ЭВМ с внутренним языком высокого уровня является SYMBOL 2R [110, 114]. Ее внутренний язык — SPL, который содержит наиболее характерные черты языков ПЛ/1, АЛГОЛ и АПЛ. Развитие архитектуры ЭВМ также повышает значимость развития методов оптимизации программ на уровне входных процедурно-ориентированных языков, так как эти методы в настоящее время и в недалеком будущем будут полезными для оптимизации программ на уровне внутренних и промежуточных языков.

Наличие более крупных конструкций во входном языке по сравнению с конструкциями промежуточного языка несет в себе некоторый элемент естественной факторизации, которая, в отличие от факторизации, выполняемой автоматически, не содержит лишних управляющих и информационных связей и в то же время сохраняет все связи, обычно теряющиеся при трансляции программ в промежуточный язык. Все большее развитие получают системы автоматизации программирования, генерирующие программы на языках высокого уровня, например на ФОРТРАНе [96] или АЛГОЛе [1]. Такие программы, как правило, требуют оптимизации. Вместе с тем для решения задачи оптимизации программ на уровне входного языка имеется хорошая основа в виде богатого теоретического наследия в области операторного метода программирования, который исследует модели программ, наиболее полно учитывающих основные особенности языков высокого уровня.

5. ПОСТАНОВКА ЗАДАЧИ ОПТИМИЗАЦИИ ПРИМЕНИТЕЛЬНО К ПРОГРАММАМ РЕАЛЬНОГО ВРЕМЕНИ

Задача оптимизации программ в общей своей постановке алгоритмически неразрешима. Поэтому основное значение имеет практический аспект решения этой задачи применительно к конкретному классу программ, свойства которых определяют характер постановки и решение рассматриваемой задачи. Одним из таких классов программ являются программы реального времени (РВ-программы), используемые в системах реального времени (РВ-системах). В связи со спецификой РВ-систем особый интерес представляет приложение задачи оптимизации к РВ-программам.

Особенности функционирования РВ-систем. Под РВ-системой понимается совокупность ЭВМ и ее программного обеспечения, в работе которых используется некоторый непрерывно и независимо от системы изменяющийся параметр внешней среды (как правило, время), явно или неявно входящий в результаты вычислений и задающий ритм работы системы [71, 77].

Работа РВ-системы инициируется событиями, происходящими во внешней среде. В ответ на эти события система должна произвести обработку некоторой информации за строго ограниченный сверху промежуток времени $T_{кр}$ и вернуть среде результаты обработки. Невыполнение работы за время $T_{кр}$ полностью или в значительной мере обесценивает ее результат и чревато нежелательными последствиями во внешней среде вплоть до существенного нарушения работы управляемого объекта, будь то технический (технологический) объект или объект экономического типа, т. е. цех, предприятие, отрасль и т. д. [76]. На практике $T_{кр}$ определяется динамическими характеристиками внешней среды и колеблется в широком диапазоне от миллисекунд до минут и более.

В качестве внешней среды обычно выступает некоторый технический объект или технологический агрегат, который часто рассматривается в совокупности с обслуживающим его персоналом. Условиям приведенного определения удовлетворяет широкий круг систем, разнящихся по производительности, объему памяти, организации вычислительного процесса, языковому обеспечению и другим важнейшим характеристикам. Действительно, мощные многомашинные наземные управляющие системы в значительной степени отличаются от бортовых систем, построенных на экономных микромашинах, и организация вычислительного процесса в системе управления химическим производством далека от организации вычислительного процесса в системе управления автоматической телефонной станцией. Согласно прогнозам, в недалеком будущем РВ-системы будут интенсивно развертываться на ЭВМ четвертого поколения.

Требования, предъявляемые к математическому обеспечению РВ-систем. Специфика РВ-систем обусловливает требования, которые приводят к значительным трудностям в разработке прикладных программ и делают весьма актуальными исследования по общим принципам их создания.

Необходимость обеспечения высокой реактивности РВ-систем при, как правило, интенсивной загрузке ЭВМ, налагает жесткие условия на время реализации управляющих алгоритмов. Все это придает специфические черты процессу программирования для РВ-систем, а именно: необходимость получения максимально высокого качества рабочих программ как по быстродействию, так и по объему занимаемой ими памяти с возможностью их корректировки. Требования высокого эффективного быстродействия и экономии памяти системы приводят к необходимости тщательной шлифовки управляющих РВ-программ, использованию различных программных ухищрений, в общем случае не нужных в управляющих программах общего назначения. Чем меньше «оболочка» отделяет технологические программы от оборудования, тем лучше. При этом следует учитывать, что базовые ЭВМ для

РВ-систем, как правило, имеют довольно простую систему команд и недостаточные аппаратные средства, что связано со стремлением сократить оборудование для повышения надежности системы, уменьшить ее габаритные размеры и массу. В частности, иногда при разработке управляющих программ приходится отказываться от удобства модульного программирования, распределения динамической памяти и т. д. Математическое обеспечение РВ-систем часто невозможно создавать на самой управляющей ЭВМ из-за ее специфических качеств: малого объема оперативной памяти, недостаточного объема внешней памяти, отсутствия ряда устройств ввода-вывода и т. д. Это приводит к использованию универсальных ЭВМ с большим быстродействием и развитым математическим обеспечением в качестве инструментальных при создании РВ-программ. Управляющие программы РВ-систем используются многократно и, как правило, функционируют в течение длительного периода времени, исчисляемого иногда годами или даже десятками лет [19], что также отражается на методах проектирования и создания программного обеспечения РВ-систем.

Необходимость предельно-достижимой оптимизации программ по времени выполнения и занимаемой памяти приводит к тому, что системы программирования для РВ-системы в настоящее время, как правило, базируются на различного типа макроассемблерах [19]. Сочетание большого объема программ РВ-систем, который, как правило, составляет десятки и сотни тысяч команд, а в настоящее время нередки случаи, когда он достигает миллионов команд [71], с производительностью труда 1—3 команды в день на человека, приводит к недопустимому увеличению сроков разработки РВ-систем, понижению эффективности их использования в период отладки и опытной эксплуатации. При растянутых сроках создания систем их моральный износ очень высок.

Технология создания математического обеспечения РВ-систем. Растянутые сроки создания РВ-систем привели к повсеместному стремлению к написанию РВ-программ на процедурно-ориентированных языках, расширенных, естественно, средствами, специфичными для РВ-программ.

Кроме того, в развитии вычислительной техники в последние годы наступил качественно новый период, характеризуемый тем, что ЭВМ начали непосредственно замыкаться на обслуживаемую ими среду, будь то технологический (технический) объект или коллектив людей (так называемых конечных пользователей). Уже сейчас пользователями средств вычислительной техники становятся лица самых разнообразных специальностей. Составление отладки и использование РВ-программ в большинстве случаев производится инженерно-техническим персоналом, работающим на соответствующем технологическом (техническом) объекте и имеющим недостаточные подготовку и опыт программирования

Эффективность участия этого инженерно-технического персонала в разработке РВ-программ в значительной мере зависит от языковых средств программирования РВ-систем. Кроме того, на сопровождение математического обеспечения (тем более математического обеспечения РВ-систем в связи с длительностью их функционирования) затрачивается 60—70 % всех средств, и эта цифра достигает максимального значения только тогда, когда система написана на языке ассемблера. Применение языков высокого уровня, безусловно, позволит снизить эти затраты, тем самым удешевив весь процесс создания программного продукта. Однако широкое применение процедурно-ориентированных языков высокого уровня для программирования задач, решаемых РВ-системами, тормозится неоптимальностью получаемых в результате трансляции объектных программ и связанным с этим перерасходом ресурсов ЭВМ, что экономически невыгодно, а в ряде случаев просто недопустимо. Например, в работе [115] приведены данные одной из типичных РВ-систем (система обработки данных летних испытаний): увеличение времени выполнения программ, оттранслированных с ФОРТРАНа, по сравнению с программами, написанными на языке ассемблера, составляет до 4,1 раза (в среднем, 2,6 раза), перерасход главной памяти для этого же случая — 2 раза. Однако развитие эффективных методов оптимизации программ, повышение уровня машинного языка, а также структурная реализация ряда рутинных функций трансляторов постепенно меняют существующее положение в пользу языков высокого уровня.

Учитывая это, а также то, что объем РВ-программ непрерывно увеличивается, можно с уверенностью сказать, что решение задачи оптимизации РВ-программ — одна из актуальных задач при создании РВ-систем, и существуют все предпосылки для успешного решения этой задачи.

Особенности задачи оптимизации программ применительно к РВ-системам. Разработка и исследование языков, ориентированных на описание алгоритмов, выполняемых в реальном масштабе времени (РВ-языков), как научное направление оформились в то время, когда уже получили значительное развитие языки, ориентированные на алгоритмы вычислительной математики, на обработку экономических данных и языки моделирования дискретных систем. Поэтому при создании РВ-языков высокого уровня, как правило, используется богатый опыт, накопленный при разработке языков смежных областей программирования.

Из анализа современных РВ-языков [78] следует, что языки, ориентированные на РВ-системы, весьма близки по содержанию и содержат две функционально различные части: мощный вычислительный аппарат, содержащий в качестве базового один из процедурно-ориентированных языков (чаще всего ФОРТРАН

или его версию), который обеспечивает собственно обрабатывающие алгоритмы, и управляющий аппарат, содержащий развитие средства программного интерфейса, включающие обращение одной ветви к другой с автоматической организацией очередей заявок и синхронизацию работы ветвей, а также синхронизацию вычислительного процесса с внешними событиями и таймером. Исходя из структуры РВ-языков высокого уровня, задачу оптимизации программ реального времени можно разделить на две части:

1. Вычислительную, являющуюся совокупностью программных модулей, записанных на языках высокого уровня. Для решения этой задачи могут быть использованы опыт и методы, разработанные для оптимизации программ, написанных на процедурно-ориентированных языках и языках ассемблера.

2. Управляющую, являющуюся по сути дела задачей оптимизации управления, а не задачей оптимизации программ. Эта задача связана с изменением алгоритма управления и решается методами теории управления.

Вопросы, рассматриваемые в настоящей книге, относятся к первой части задачи оптимизации РВ-программ. При решении этой задачи необходимо учитывать следующую специфику РВ-систем. Число типов управляющих ЭВМ, разработанных к настоящему времени, велико и продолжает увеличиваться. При этом некоторые машины устаревают, их заменяют новыми. Унификация этих ЭВМ пока не представляется возможной, так как их внутренняя структура ориентирована на решение определенного типа задач [19]. Проблемы переноса программного обеспечения, в том числе и систем оптимизации программ, с ЭВМ на ЭВМ требуют значительных усилий, то же самое можно сказать о попытках его переноса на новую операционную систему для той же ЭВМ. Вследствие разнообразия типов управляющих ЭВМ это, в общем случае, определяет большую стоимость решения задачи оптимизации программ для РВ-систем в полном объеме (т. е. создания систем оптимизации программ для всех типов РВ-систем), так как каждая создаваемая система — это уникальная разработка, а это не позволяет поставить процесс оптимизации программ на «промышленную основу» [27].

Требования функционирования ЭВМ в режиме реального масштаба времени и, как правило, большая стоимость компонент таких ЭВМ определяет простоту системы команд (например, отсутствие регистров и многоуровневой памяти).

Перечисленная специфика РВ-систем определяет архитектуру и особенности построения систем оптимизации РВ-программ. Широкий диапазон решаемых задач, как правило, разовой постановки, и постоянная замена и модификация управляющих ЭВМ обусловливают постоянно меняющееся множество входных языков РВ-систем и позволяют говорить об открытости систем

оптимизации РВ-программ и, следовательно, их параметрической настройки на входной язык (см. гл. 3.4). Отметим, что оптимизация РВ-программ на уровне входного процедурно-ориентированного языка обладает (наряду с некоторыми из рассмотренных в гл. 3.4 свойств) рядом важных свойств, позволяющих ей занять особое место в процессе оптимизации РВ-программ:

1. Оптимизирующий блок, выполняющий преобразование программ на уровне процедурно-ориентированного языка, является общим блоком для широкого класса управляющих ЭВМ, и поэтому создание оптимизирующего транслятора для конкретной управляющей ЭВМ сводится к созданию оптимизирующих блоков, выполняющих преобразования программ на уровне промежуточного языка и /или языка ассемблера. Кроме того, наличие мощных оптимизирующих трансляторов на уровне входных языков позволяет создавать трансляторы с таких языков в машинные коды, которые основаны на заведомо упрощенных алгоритмах трансляции, учитывающих предварительную оптимизацию. Перечисленные факторы позволяют существенно снизить как сроки разработки отдельных оптимизирующих трансляторов, так и суммарную стоимость создания оптимизирующих трансляторов для всего парка управляющих ЭВМ.

2. Благодаря простоте системы команд, управляющих ЭВМ, и, как следствие, отсутствию необходимости оптимизации использования общих регистров и разных уровней памяти оптимизация РВ-программ на уровне процедурно-ориентированного языка является более важной и обладает более высокими возможностями, чем оптимизация на том же уровне простых вычислительных программ, решаемых на мощных ЭВМ, для которых существенное значение имеет оптимизация распределения быстрых регистров.

3. РВ-программы, прошедшие этап оптимизации на уровне входного языка, сохраняют свою наглядность не только для разработчика этой программы, но и для других программистов, что облегчает решение кадровых организационных вопросов при передаче программ от одного программиста другому.

4. Оптимизация программ на уровне входного языка позволяет легко осуществлять их отладку и без большого труда вносить в их текст изменения и дополнения без необходимости повторной оптимизации этой программы.

Анализ состава команд РВ-программ [71] показал, что по сравнению с вычислительными программами они содержат существенно больше глобальных переменных и команд перехода (условного и безусловного), т. е. структура управляющего и информационного графов таких программ сложнее структуры аналогичных графов вычислительных программ. Усложненная структура управляющего и информационного графов РВ-программ вместе с требованием получения предельно достижимой оптими-

зации обусловливают развитие методов оптимизации программ, позволяющих учитывать максимальное число управляющих и информационных связей между переменными и операторами программ, т. е. методов глобальной оптимизации программ.

Так как функциональные программы в РВ-системах выполняются многократно, их оптимизацию можно довести до высокого уровня, благодаря тому, что время трансляции (с учетом выполнения оптимизационных алгоритмов) практически ничтожно по сравнению с суммарным временем работы этих программ, т. е. в этом случае оправдано применение даже наиболее сложных оптимизационных алгоритмов, что позволяет выполнить глобальную оптимизацию РВ-программ. Кроме того, использование универсальных ЭВМ с большим быстродействием, имеющих развитое математическое обеспечение, в качестве инструментальных средств при создании РВ-программ служит предпосылкой для реализации даже наиболее сложных оптимизационных алгоритмов.

6. ДВЕ СИСТЕМЫ ОПТИМИЗАЦИИ

К настоящему времени в нашей стране создано несколько оптимизирующих трансляторов и систем оптимизации. Одним из первых оптимизирующих трансляторов, для которого была разработана машинная операция, позволяющая существенно сократить число логических операций, осуществляющих динамические пересылки, является ТА [100]. Ряд преобразований удачно реализован в оптимизирующем варианте транслятора ФОРТРАН-Дубна [80], входящего в состав математического обеспечения ЭВМ БЭСМ-6. Широкое использование этого транслятора является источником информации, достаточной для оценки эффективности реализации ряда оптимизационных процедур, в том числе и чистки циклов [79]. Интересным и практичным является подход к оптимизации программ, принятый в многоязыковой системе программирования БЕТА [81], основанный на том, что выбранные оптимизационные преобразования являются полезными и могут быть реализованы за приемлемую цену. Хотя в каждой конкретной системе оптимизации программ (оптимизирующем трансляторе) реализованы или реализуются лишь немногие из перечисленных возможностей, в совокупности они составляют то множество приемов, стратегий, подходов, методов, описанных ранее. Наиболее полно рассмотренные возможности реализованы в системе АЛЬФА-6 [89] и оптимизирующем трансляторе с языка ПЛ/1 операционной системы ЕС ЭВМ. Здесь кратко описаны основные особенности этих систем.

Система АЛЬФА-6 представляет собой оптимизирующий транслятор, созданный для ЭВМ БЭСМ-6 и мониторной систе-

мы Дубна [89]. Входной язык системы является расширением языка АЛГОЛ-60 и называется альфа-6. Назначение системы АЛЬФА-6 — генерация высококачественных объектных программ, что обеспечивает ее применение для решения задач с предельными характеристиками. Оттранслированная системой АЛЬФА-6 программа представляет собой модуль загрузки мониторной системы Дубна. Можно выделить следующие основные компоненты системы АЛЬФА-6: языки системных команд альфа-6 и язык данных; транслятор; редакторы поперфокартный и текстовый; отладчик; программа библиотечных работ; библиотека стандартных процедур; препроцессор. Основными добавлениями, по отношению к языку АЛГОЛ-60, являются: стандартные операторы языка АЛГАМС; возможности записи команд БЭСМ-6 с мнемоническими обозначениями автокода БЕМШ и средства комплексации раздельно транслируемых модулей в рамках мониторной системы Дубна. Система выполняет следующие основные работы.

Трансляцию программы, заключающуюся в переводе исходной программы, записанной на входном языке альфа-6, в эквивалентную ей программу, записанную на языке модуля загрузки мониторной системы Дубна.

Подготовку данных, заключающуюся в переводе исходных альфа-данных в данные, воспринимаемые мониторной системой Дубна.

Редактирование (поперфокартное и текстовое) программы или данных, заключающееся в изменении текста программы или данных в соответствии с правилами редактирования в системе АЛЬФА-6. Отладочное редактирование состоит во вставке в исходную альфа-программу некоторых операторов, которые помогают программисту осуществлять отладку программы после ее трансляции.

Печать и перфорацию, позволяющую выдавать на АЦПУ или на перфокарты программу, данные и модуль загрузки (частично или полностью).

Библиотечные работы, производящие следующие операции с архивами системы АЛЬФА-6: создание архива; запись программы или данных в архив; уничтожение программы или данных из архива; печать оглавления архива; изменение имени архива.

Во время трансляции в программе производятся оптимизационные преобразования (над программой выполняются оптимизационные процедуры). Наиболее важными из них являются: применение «смешанной» стратегии для трансляции формальных и фактических параметров процедур; чистка циклов; оптимизация выражений; локальная экономия памяти; глобальная экономия памяти. При оптимизации учитывается система команд ЭВМ БЭСМ-6. Например, если параметры циклов и переменные с индексами в теле цикла часто программируются с использова-

нием индексных регистров, то значение скаляра-параметра цикла в отведенной для него ячейке памяти может не соответствовать действительному значению этого скаляра [89].

Средства отладки включают в себя: трассировку программы, контроль за выполнением программы, сбойную распечатку. Трассировку программы можно осуществлять печатаньем значений указываемых переменных или меток выполняемых помеченных операторов. Контроль за выполнением программы представляет собой контроль соблюдения границ массивов; локализацию места аварийного останова или контроль соответствия формальных и фактических параметров процедур по виду, типу и структуре.

При обнаружении ошибки во время выполнения альфа-программы может произойти прерывание. В этом случае на АЦПУ выдается сбойная распечатка, содержащая достаточно подробную информацию о программе.

Оптимизирующий транслятор с языка ПЛ/1 ОС ЕС ЭВМ создан для трансляции программ, записанных на языке ПЛ/1, в эффективный код. Транслятор требует как минимум 50 Кбайт ОЗУ на ЭВМ серии ЕС со стандартной системой команд, и предполагается, что он удовлетворяет всем требованиям пользователей, предъявляемым к языку ПЛ/1 при использовании ЭВМ серии ЕС среднего или большого размера. Рассматриваемый транслятор включает следующие дополнительные средства языка ПЛ/1: оператор DEFAUT; переменные типов файл (FILE) и вход (ENTRY). Основная цель транслятора — генерирование объектных программ, которые должны выполняться как можно быстрее и занимать как можно меньше памяти во время выполнения. В одних случаях выполняется генерация эффективного кода для последовательностей предложений, написанных программистом, в других случаях транслятор может изменять последовательность предложений для того, чтобы усовершенствовать программу без изменения результата ее выполнения. Если дано указание транслятору для осуществления оптимизации, то для получения эффективной объектной программы над генерируемым объектным кодом будут выполнены все оптимизационные процедуры. Если не будет дано указаний транслятору для выполнения оптимизации, то время трансляции, естественно, будет меньше. Транслятор выполняет следующие оптимизационные процедуры: оптимизацию выражений; чистку циклов; использование эффективных распознавателей; упрощение выражений; сокращение числа вычислений индексных функций; устранение общих констант; выполнение константных действий; замену возведения в степень и умножения; использование конструкций DATA и in-line кодов для преобразования типов данных, вызова операторов ввода-вывода, объединения списков формата и операторов вывода, обработки строковых данных и для встроенных

функций; оптимизацию через «щель» и использование быстрых регистров; уменьшение размерности обрабатываемых массивов.

Скорость трансляции зависит от конфигурации ЭВМ, размещения данных, объема разделов, сложности программы, средств языка и используемых опций транслятора. В общем случае быстрая трансляция достигается при большом объеме памяти или разделе и минимальном использовании опций транслятора. При выполнении оптимизации время трансляции увеличивается в среднем на 25 %, а в некоторых случаях может достигать 100 %.

При выполнении оптимизации время выполнения генерируемого транслятором объектного кода может быть значительно меньше времени выполнения объектного кода, генерируемого транслятором с языка ПЛ/1 уровня F. При испытании транслятора суммарное время выполнения научно-ориентированных программ, над которыми была выполнена оптимизация, было на 50 % меньше, чем время выполнения тех же программ, транслированных транслятором с языка ПЛ/1 уровня F, колеблясь в диапазоне от 0 % до 63 % для отдельных программ. Суммарное время выполнения ряда коммерчески-ориентированных программ, над которыми была выполнена оптимизация, было на 30 % меньше, чем время выполнения тех же программ, транслированных транслятором для языка ПЛ/1 уровня F, колеблясь для отдельных программ от 0 до 59 %. Наименьшее повышение эффективности после оптимизации как для научных, так и коммерческих программ, как правило, получено для самых коротких программ.

Диагностические средства применяются как в процессе трансляции, так и в процессе выполнения. При обработке предложений программы, содержащих ошибки, транслятор может пытаться исправить ошибки, делая предположения о правильном виде предложений. Сообщения об ошибках, выдаваемые во время трансляции, делятся на категории: непоправимые ошибки, грубые ошибки, замечания и информационные сообщения. Сообщения печатаются, группируясь по категориям в порядке номеров предложений, содержащих эти ошибки. Каждое сообщение содержит число предложений с ошибками, или, когда это можно оценить, часть предложения, содержащего ошибку; некоторые сделанные предложения и действия, предпринятые транслятором.

По приведенным кратким описаниям двух трансляторов (систем) хорошо видна качественная разница между первыми оптимизирующими трансляторами и современными системами оптимизации программ, состоящая не только в увеличении оптимизационных возможностей, но и в расширении сервиса, предоставляемого программисту, такого сервиса, который существенно меняет весь процесс создания программы.

СПИСОК ЛИТЕРАТУРЫ

1. **Автоматизация** построения программ для расчета схем теплоэнергетических установок / В. Г. Карпов, Л. С. Попырин, В. И, Самусев, В. В. Эпельштейн.— Изв. АН СССР. Сер. Энергетика и транспорт, 1973, № 1, с. 129—138.

2. **АЛЬФА-система** автоматизации программирования / Под ред. А. П. Ершова. Новосибирск : СО АН СССР, 1965. 263 с.

3. **Архангельский Б. В.** Алгоритм выделения в операторных схемах областей постоянства переменных.— Программирование, № 4, 1979, с. 16—28.

4. **Архангельский Б. В.** Возможности повышения качества программ, написанных на ФОРТРАНе. К., 1976. 27с. (Препринт ИК АН УССР; № 36).

5. **Архангельский Б. В.** Некоторые оптимизационные процедуры над оперативными схемами.— Программирование, 1980, № 3, с. 13—27.

6. **Архангельский Б. В.** Некоторые стратегии и методы оптимизации программ.— В кн.: Математическое обеспечение ВЦКП. Сер. Надежность и качество программного обеспечения К. : ИК АН УССР, 1981, с. 3—20.

7. **Архангельский Б. В.** Обзор оптимизационных процедур.— УСиМ, 1981, № 4, с. 103—109.

8. **Архангельский Б. В.** Оптимизация программ на основе преобразования структуры программы и устранения избыточных вычислений.— УСиМ, 1982, № 5, с. 35—41.

9. **Архангельский Б. В., Никитин А. И.** Вопросы оптимизации программ, записанных в процедурно-ориентированных языках для систем реального времени.— УСиМ, 1975, № 6, с. 6—11.

10. **Архангельский Б. В., Никитин А. И.** Об одном алгоритме построения областей постоянства переменных.— Кибернетика, 1976, № 4, с. 134—137.

11. **Архангельский Б. В., Никитин А. И.** Об одном алгоритме реализации процедуры оптимизации выражений на уровне входного языка.— УСиМ, 1976, № 5, с. 35—40.

12. **Архангельский Б. В., Никитин А. И.** О возможности автоматической оптимизации программ в системах реального времени.— УСиМ, 1976, № 2, с. 48—53.

13. **Архангельский Б. В., Никитин А. И., Черняховский В. В.** Устойчивые типы семантических ошибок программ.— В кн.: Тез. докл. Всесоюз. конф. «Синтез, тестирование, верификация и отладка программ». Рига : Латв. ун-т, 1981, с. 17.

14. **Ахо А., Ульман Дж.** Теория синтаксического анализа перевода и компиляции. Т. 2 / Пер. с англ.; Под ред. В. М. Курочкина. М. : Мир, 1978. 487 с.

15. **Бабаян Б. А.** Основные принципы программного обеспечения МВК «Эльбрус». Препринт № 5, ИТМ и ВТ АН СССР, 1977. 11 с.

16. **Берж К.** Теория графов и ее применение. М. : Изд-во Иностр. лит., 1962. 320 с.

17. **Бурцев В. С.** Принципы построения многопроцессорных вычислительных комплексов «Эльбрус». М., 1977. 53 с. (Препринт ИТМ и ВТ АН СССР; № 1).

18. **Васючкова Т. С.** Оценка качества реализации языковых конструкций в трансляторах.— Программирование, 1980, № 5, с. 63—72.

19. **Виленкин С. Я., Трахтенгерц Э. А.** Математическое обеспечение управляющих вычислительных машин. М. : Энергия, 1972. 392 с.

20. **Гейдан Л. И., Скопин И. Н., Черевикин А. Ю.** О метрологическом исследовании программ.— В кн.: Тез. докл. 1-й Всесоюз. конф. «Технология программирования», Секция К. К. : ИК АН УССР, 1979, с. 12—13.

21. **Герасименко В. А.** Основные принципы структурного программирования.— Зарубеж. радиоэлектроника, 1976, № 11, с. 3—10.

22. **Гладкий А. В.** Формальные грамматики и языки. М. : Наука, 1973, с. 368.

23. **Глушков В. М.** Диалог с вычислительной машиной : Современные возможности и перспективы.— УСиМ, 1974, № 1, с. 3—7.

24. **Глушков В. М.** Теория автоматов и вопросы проектирования структур цифровых машин.— Кибернетика, 1965, № 1, с. 3—11.

25. **Глушков В. М.** Теория автоматов и формальные преобразования микропрограмм.— Кибернетика, 1965, № 5, с. 1—10.

26. **Глушков В. М., Летичевский А. А.** Теория дискретных преобразователей.— В кн.: Избранные вопросы алгебры и логики. Новосибирск : Наука, 1973, с. 5—39.

27. **Глушков В. М., Вельбицкий И. В., Стогний А. А.** Об одном подходе к построению системного математического обеспечения современных вычислительных машин.— Кибернетика, 1972, № 3, с. 25—35.

28. **Глушков В. М., Капитонова Ю. В., Летичевский А. А.** О применении метода формализованных технических заданий к проектированию программ обработки структур данных.— Программирование, 1978, № 6, с. 31—43.

29. **Глушков В. М., Никитин А. И., Рабинович З. Л.** Некоторые тенденции развития структур и программного обеспечения ЦВМ.— УСиМ, 1972, № 1, с. 79—85.

30. **Глушков В. М., Цейтлин Г. Е., Ющенко Е. Л.** Алгебра. Языки. Программирование. К. : Наук. думка, 1978. 320 с.

31. **Глушков В. М., Цейтлин Г. Е., Ющенко Е. Л.** Методы символьной мультиобработки. К. : Наук. думка, 1980. 242 с.

32. **Глушков В. М., Цейтлин Г. Е., Ющенко Е. Л.** Многоуровневое структурное проектирование программ : Формализация метода — сфера приложения.— Кибернетика, 1981, № 4, с. 42—65.

33. **Глушков В. М., Цейтлин Г. Е., Ющенко Е. Л.** Проблемы анализа и синтеза структурированных параллельных программ.— Кибернетика, 1981, № 3, с. 1—16.

34. **Горелик А. М.** Анализ некоторых способов сравнения качества трансляторов.— Программирование, 1981, № 1, с. 79—82.

35. **Грис Д.** Конструирование компиляторов для цифровых вычислительных машин / Пер. с англ.; Под ред. Ю. М. Баяковского и В. С. Штаркмана. М: Мир, 1975. 544 с.

36. **Дал У, Дейкстра Э., Хоор К.** Структурное программирование / Пер. с англ.; Под ред. Э. З. Любимского, В. В. Мартынюка. М. : Мир, 1975. 245 с.

37. **Дейкстра Э.** Дисциплина программирования / Пер. с англ.; Под ред. Э. З. Любимского. М. : Мир, 1978. 275 с.

38. **Дзержинский Ф. Я.** ПСЕВДОКОД (язык проектирования программ) и правила структурного программирования М. : ЦНИИатоминформ., 1979. 58 с.

39. **Дзержинский Ф. Я., Тер-Сааков А. И.** Технология программирования — структурный подход. М. : ЦНИИавтоминформ., 1978. 87 с.

40. **Ершов А. П.** Введение в теоретическое программирование. М. : Наука, 1977. 288 с.

41. **Ершов А. П.** Об одном теоретическом принципе системного программирования.— Докл. АН СССР, 1977, т. 233, № 2, с. 272—275.

42. **Ершов А. П.** О сущности трансляции.— Программирование, 1977, № 5, с. 21—39.

43. **Ершов А. П.** Проектные характеристики многоязыковой системы программирования.— Кибернетика, 1975, № 4, с. 11—27.

44. **Ершов А. П.** Сведение задачи распределения памяти при составлении программ к задаче раскраски вершин графов.— Докл. АН СССР, 1962, т.142, № 4, с. 785—787.

45. **Ершов А. П.** Современное состояние схем программ.— Проблемы кибернетики, 1973, № 27, с. 87—110.

46. **Ершов А. П., Ляпунов А. А.** О формализации понятия программы.— Кибернетика, 1967, № 5, с. 40—57.

47. **Задыхайло И. Б.** Алгоритм выделения всех вершин ориентированного графа, принадлежащих контурам.— Цифровая вычислител. техника и программирование, 1967, № 3, с. 166—172.

48. **Закревский А. Д., Поляков А. М.** О повышении качества машинных программ путем оптимизации использования регистров.— УСиМ, 1976, № 6, с. 105—111.

49. **Зыков А. А.** О некоторых свойствах линейных комплексов.— Мат. сб. М. : Физматгиз, 1949, вып. 2, с. 163—183.

50. **Ингерман П.** Синтаксически-ориентированный транслятор. М. : Мир, 1969. 174 с.

51. **Интерактивная** обработка структур данных / В. В. Бублик, А. Е. Дорошенко, С. Л. Кривой, В. Ф. Лябах — Кибернетика, 1978, № 5, с. 13—18.

52. **Исследование** вычислительного процесса на ЭВМ типа «Промінь» и пути повышения эффективного быстродействия этого класса машин / Ю. В. Благовещенский, Б. А. Попов, И. В. Сергиенко, Т. С. Теслер — Кибернетика, 1970, № 3, с. 54—58.

53. **Йодан Э.** Структурное проектирование и конструирование программ : Пер. с англ. М. : Мир, 1979.— 416 с.

54. **Касьянов В. Н.** Анализ структур программ.— Кибернетика, 1980, № 1, с. 48—61.

55. **Касьянов В. Н.** Анализ управляющих графов программ.— В кн.: Системное программирование. Ч. 2. Новосибирск : ВЦ СО АН СССР, 1973, с. 35—48.

56. **Касьянов В. Н.** К вопросу о реализации схем над распределенной памятью.— Кибернетика, 1977, № 1, с. 63—68.

57. **Касьянов В. Н.** О нахождении аргументов и результатов операторов в схемах с косвенной адресацией.— Программирование, 1967, № 1, с. 6—15.

58. **Касьянов В. Н.** Смешанные вычисления и оптимизация программ.— Кибернетика, 1980, № 2, с. 51—54.

59. **Касьянов В. Н., Поттосин И. В.** Технологические возможности оптимизации программ.— Программирование, 1980, № 2, с. 27—30.

60. **Котов В. Е.** Ввведение в теорию схем программ. Новосибирск : Наука, 1978. 256 с.

61. **Кривой С. Л.** О поиске инвариантных соотношений в программах.— В кн.: Математическая теория проектирования вычислительных машин. К. : ИК АН УССР, 1978, с. 35—51.

62. **Кривой С. Л.** О поиске инвариантных соотношений в программах над абелевыми группами данных.— В кн.: Методы проектирования схемного и программного оборудования ЭВМ. К. : ИК АН УССР, 1979, с. 25—32.

63. **Кривой С. Л.** Поиск соотношений в программах под свободными группами данных.— В кн.: Средства автоматизированного проектирования вычислительных систем. К. : ИК АН УССР, 1979, с. 25—32.

64. **Криницкий Н. А.** Равносильные преобразования алгоритмов. М. : Сов. радио, 1970. 304 с.

65. **Лавров С. С.** Об экономии памяти в замкнутых операторных схемах.— ЖВМ и МФ, 1961, т. 1, № 4, с. 687—701.

66. **Летичевский А. А.** Об ускорении итерации монотонных операторов.— Кибернетика, 1976, № 4, с. 1—7.

67. **Летичевский А. А.** Эквивалентность автоматов относительно полугрупп.— Теорет. кибернетика, 1970, № 6, с. 72.

68. **Летичевский А. А.** Эквивалентность и оптимизации программ.— В кн.: Тр. междунар. симпоз. «Теория программирования». Ч. 1. Новосибирск : ВЦ СО АН СССР, 1972, с. 166—180.

69. **Летичевский А. А., Годлевский А. Б.** Оптимизация алгоритмов в процессе их проектирования методом формализованных технических задач.— В кн.: Автоматизация проектирования ЭВМ и их компонентов. К. : ИК АН УССР, 1977, с. 46—71.

70. **Липаев В. В.** Проектирование математического обеспечения АСУ : Системотехника, архитектура, технология. М. : Сов. радио, 1977. 400 с.

71. **Липаев В. В., Колин К. К., Серебровский Л. А.** Математическое обеспечение управляющих ЦВМ. М. : Сов. радио, 1972. 528 с.

72. **Льюис Ф., Розенкранц Д., Стирнз Р.** Теоретические основы проектирования компиляторов / Пер. с англ.; Под ред. В. Н. Агафонова. М. : Мир, 1979. 582 с.

73. **Малиновский В. Н., Белецкий Р. И.** Оптимальное размещение микропрограмм в страничной микропрограммной памяти.— УСиМ, 1978, № 5, с. 64—68.

74. **Мартынюк В. В.** Об изменении порядка выполнения операторов в операторной схеме.— Цифровая вычислител. техника и программирование, 1967, № 2, с. 94—107.

75. **Мартынюк В. В.** Об экономном распределении памяти.— ЖВМ и МФ, 1962, т. 2, № 3, с. 445—458.

76. **Никитин А. И.** О функциональных возможностях операционных систем реального времени.— УСиМ, 1973, № 5, с. 26—33.

77. **Никитин А. И.** Общее программное обеспечение систем реального времени. К. : Наук. думка, 1980. 132 с.

78. **Никитин А. И., Шурубура В. П.** Об особенностях входных языков для вычислительных систем, работающих в реальном масштабе времени.— УСиМ, 1973, № 4, с. 26—33.

79. **О Ен Ир, Шириков В. П.** Об одном оптимизирующем варианте ОРТ-1 обработки циклов при трансляции с языка ФОРТРАН на ЭВМ БЭСМ-6. Дубна: ОИЯИ, 1975, с. 50—56.

80. **О Ен Ир, Шириков В. П.** Об оптимизирующих вариантах транслятора с языка ФОРТРАН для ЭВМ БЭСМ-6.— Тр. конф. «Проблемы повышения эффективности БЭСМ-6». Иркутск, 1976, с. 50—56.

81. **Поттосин И. В.** Глобальная оптимизация практический подход.— В кн.: Тр. Всесоюз. симпоз. по методам реализации новых алгоритмических языков. Ч. 1. Новосибирск, 1970, с. 113—128.

82. **Поттосин И. В.** К задаче чистки циклов.— Цифровая вычислительная техника и программирование. М. : Сов. радио, 1968, вып. 4, с. 18—36.

83. **Поттосин И. В.** К обоснованию алгоритмов оптимизации программ.— Программирование, 1979, № 2, с. 3—13.

84. **Поттосин И. В.** О линеаризации программы и частичном ее упорядочении.— В кн.: Системное и теоретическое программирование. Новосибирск : ВЦ СО АН СССР, 1972, с. 278—286.

85. **Поттосин И. В., Югринова О. В.** Обоснование преобразования чистка циклов.— Программирование, 1980, № 5, с. 3—16.

86. **Рау О. И.** Модель виртуальной машины как база оптимизации программ.— В кн.: Вопросы системного программирования. М. : МГУ, 1978, с. 75—90.

87. **Сабельфельд В. К.** Реализация процедур в многоязыковом транслято-ре.— Тр. Всесоюз. симпоз. по методам реализации новых алгоритмических языков. Ч. 1. Новосибирск : ВЦ СО АН СССР, 1975, с. 129—142.

88. **Салтыков А. И., Макаренко Г. И.** Программирование на языке ФОРТРАН. М. : Наука, 1976. 255 с.

89. **Транслятор** АЛЬФА-6 в системе Дубна / Под ред. А. П. Ершова. М. : Наука, 1979. 352 с.

90. **Трахтенброт М. Б.** О преобразованиях, доопределяющих программу.— Кибернетика, 1980, № 2, с. 55—60.

91. **Уичман Б. А.** Некоторые аспекты эффективности языков системного программирования.—В кн.: Тр. рабочей конф. Междунар. федерации по обраб. информации Т. 1. Новосибирск : ВЦ СО АН СССР, 1978, с. 57—73.

92. **Хьюз Дж., Мичтом Дж.** Структурный подход к программированию / Пер. с англ.; Под ред. В. Ш. Кауфмана. М. : Мир, 1980. 278 с.

93. **Цагельский В. И.** Система программирования ФОРТРАН для многопрограммной ЭВМ широкого применения : Автореф. дис. ... канд. физ.-мат. наук. Минск, 1976. 28 с.

94. **Цейтлин Г. Е.** Многоуровневое параллельное проектирование структурированных программ.— В кн.: Методы математической логики в проблемах искусственного интеллекта. Вильнюс. : ИМК АН ЛитССР, с. 262—264.

95. **Цейтлин Г. Е.** Проблема тождественных преобразований схем структурированных программ с замкнутыми логическими условиями.— Кибернетика, 1978, № 3, с. 50—57; № 4, с. 10—18; 1979, № 5, с. 44—51.

96. **Цымбал Г. Я.** Разработка методов и комплексов программ для автоматизации построения вычислительного процесса при решении расчетных задач инженерной подготовки производства : Автореф. дис. ... канд. физ. мат. наук. Минск, 1971. 25 с.

97. **Штаркман В. С.** Блок экономии рабочих ячеек в ПП-2.— Пробл. кибернетики, 1958, вып. 1, с. 185—189.

98. **Штаркман В. С.** Сравнительный анализ транслятора FOREX. М., 1978. 56 с. (Препринт, ИПМ АН СССР; № 129).

99. **Штаркман В. С., Кузнецова Т. П.** Результаты статистического исследования программ на М-20.— Вопр. техн. эксплуатации вычислител. машин, 1967, вып. 1, с. 54—75.

100. **Шура-Бура М. Р., Любимский Э. З.** Транслятор АЛГОЛ-60.— ЖВМ и МФ, 1964, № 1, с. 96—112.

101. **Шура-Бура М. Р., Мартынюк В. В.** Об эффективной организации динамического использования памяти.— ЖВМ и МФ, 1964, т. 4, № 5, с. 963—967.

102. **Allen F. E.** Program Optimisation — Annual Review in Automated Programming. Pergamon Press. V. 5., 1969, p. 1—69.

103. **Busam Vincent A., Englund Donald E.** Optimisation of Expressions in FORTRAN — Comm. ACM, 1969, N 2, p. 666—674.

104. **Chis Larson.** The Efficient Use of FORTRAN. Datamation, 1971, N T—15, p. 24—31.

105. **Corrado Bohm, Guiseppe Jacopini.** Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules.— Comm. ACM, 1966, V. 9. May, p. 366—371.

106. **David B. Loveman.** Program Improvement by Sourse to Sourse Transformation.— J. ACM, 1977, V. 24, N 1, p. 121—145.

107. **Earnest C. P., Balke K. G., Anderson J.** Analasis of Graphs by Ordering of Nodes.— J. ACM., 1972, V. 19, N 1, p. 23—42.

108. **Harlan D. Milles.** Mathematical Foundations for Structured Programming.— Report FSC72—6012, IBM Corporation, Gaithersburg, 1972, p. 2—40.

109. **Hech M. S.** Topological Sorting and Flow Graphs.— Proc. IFIP Congress 74, Stocholm, : Pub. Co., 1974, p. 125—129.

110. **Hutchinson P. C.** Ethington K. Program Execution in the SYMBOL 2R Computer.— ACM — IEEE. Symposium on High — Level — Language Computer Architecture. New York. : IEEE, 1973, p. 20—25.

111. **Irons E. T.** Asyntax Directed Compiler for Algol — 60.— Comm. ACM, 1961, N 1, p. 51—55.

112. **Knuth D. E.** An Empirical Study of FORTRAN Programs.— Software — Practice I Experience, 1971, V. 1, N 2, p. 105—133.

113. **Landin P. J.** The Next 700 Programming Languages.— Comm. ACM, 1966, N 3, p. 157—164.

114. **Richards H., Wright C.** Introduction to the SYMBOL 2R Programming Language.— ACM — IEEE Symposium on Hight — Level — Language Computer Architecture, New York. : IEEE, 1973, p. 27—33.

115. **Rymer J. W.** Waht should real time tosting offer.— IAAA Paper, 1977, № 783, p. 78—83.

116. **Schneck P. B., Evinor Angel.** A FORTRAN to FORTRAN optimising Compiler.— The Computer Jourual, 1974, V. 16, N 4, p. 322—330.

117. **Wilner W. T.** Design of the Burrought B 1700.— Proceedings of the 1972, Fall Joint Computer Conference. New York. AFIPS Press, 1977, p. 489—497.

118. **Wirth N.** Program development by stepwise refinement.— Comm. ACM, 1971, V. 14, N 4, p. 221—227.

119. **Wulf W. A., Russell D. B., Haberman A. M.** BLISS a language for systems programming.— Comm. ACM, 1971, V. 14, N 12, p. 780—790.

Стр.

Введение 3

Глава 1. Задача оптимизации программ

1. Общие понятия 6

2. Уровни языков, на которых выполняются оптимизационные преобразования 14

3. Области оптимизации 19

Глава 2. Оптимизационные процедуры

1. Особенности описания оптимизационных процедур 29

2. Оптимизация циклов 30

3. Оптимизация индексных выражений 42

4. Оптимизация логической структуры программы 49

5. Устранение избыточных вычислений 60

6. Эффективная организация подпрограмм 75

7. Использование эффективных операций 88

8. Оптимизация ввода-вывода 98

9. Эффективное использование памяти 100

10. Вспомогательные процедуры 104

Глава 3. Практические аспекты оптимизации программ

1. Методы оптимизации программ 108

2. Этапы технологии производства программ 120

3. Структурный подход к программированию 127

4. Структура систем оптимизации программ 143

5. Постановка задачи оптимизации применительно к программам реального времени 152

6. Две системы оптимизации 158

Список литературы 162

ОГЛАВЛЕНИЕ

Стр.

Введение 3

Глава 1. Задача оптимизации программ

1. Общие понятия 6

2. Уровни языков, на которых выполняются оптимизационные преобразования 14

3. Области оптимизации 19

Глава 2. Оптимизационные процедуры

1. Особенности описания оптимизационных процедур 29

2. Оптимизация циклов 30

3. Оптимизация индексных выражений 42

4. Оптимизация логической структуры программы 49

5. Устранение избыточных вычислений 60

6. Эффективная организация подпрограмм 75

7. Использование эффективных операций 88

8. Оптимизация ввода-вывода 98

9. Эффективное использование памяти 100

10. Вспомогательные процедуры 104

Глава 3. Практические аспекты оптимизации программ

1. Методы оптимизации программ 108

2. Этапы технологии производства программ 120

3. Структурный подход к программированию 127

4. Структура систем оптимизации программ 143

5. Постановка задачи оптимизации применительно к программам реального времени 152

6. Две системы оптимизации 158

Список литературы 162

ОГЛАВЛЕНИЕ

Стр.

Введение 3

Глава 1. Задача оптимизации программ

1. Общие понятия 6

2. Уровни языков, на которых выполняются оптимизационные преобразования 14

3. Области оптимизации 19

Глава 2. Оптимизационные процедуры

1. Особенности описания оптимизационных процедур 29

2. Оптимизация циклов 30

3. Оптимизация индексных выражений 42

4. Оптимизация логической структуры программы 49

5. Устранение избыточных вычислений 60

6. Эффективная организация подпрограмм 75

7. Использование эффективных операций 88

8. Оптимизация ввода-вывода 98

9. Эффективное использование памяти 100

10. Вспомогательные процедуры 104

Глава 3. Практические аспекты оптимизации программ

1. Методы оптимизации программ 108

2. Этапы технологии производства программ 120

3. Структурный подход к программированию 127

4. Структура систем оптимизации программ 143

5. Постановка задачи оптимизации применительно к программам реального времени 152

6. Две системы оптимизации 158

Список литературы 162

Борис Владимирович Архангельский, канд. физ.-мат. наук
Андрей Иванович Никитин, д-р техн. наук

СИСТЕМЫ ОПТИМИЗАЦИИ ПРОГРАММ

Редактор *О. П. Веремейчик*
Оформление художника *А. А. Белянина*
Художественные редакторы *Л. А. Дикарев, В. С. Шапошников*
Технический редактор *Е. М. Гороховская*
Корректор *В. Н. Руденко*

Информ. бланк № 2071

Сдано в набор 16.11.82. Подписано в печать 28.07.83. БФ 07193. Формат 84×108 1/32. Бумага типогр. № 1. Гарн. лит. Печ. выс. Усл. печ. л. 8,82. Усл. кр.-отт. 9,14. Уч.-изд. л. 10,48. Тираж 4000 экз. Зак. № 2-332. Цена 90 к.

Издательство «Техника». 252601. Киев. 1. Крещатик, 5.

Отпечатано с матриц Головного предприятия республиканского производственного объединения «Полиграфкнига». 252057. Киев. ул. Довженко. 3 на книжной фабрике «Жовтень». 252053. Киев. Артема, 25.

Борис Владимирович Архангельский, канд. физ.-мат. наук
Андрей Иванович Никитин, д-р техн. наук

СИСТЕМЫ ОПТИМИЗАЦИИ ПРОГРАММ

Редактор *О. П. Веремейчик*
Оформление художника *А. А. Белянина*
Художественные редакторы *Л. А. Дикарев, В. С. Шапошников*
Технический редактор *Е. М. Гороховская*
Корректор *В. Н. Руденко*

Информ. бланк № 2071

Сдано в набор 16.11.82. Подписано в печать 28.07.83. БФ 07193. Формат 84×108 1/32. Бумага типогр. № 1. Гарн. лит. Печ. выс. Усл. печ. л. 8,82. Усл. кр.-отт. 9,14. Уч.-изд. л. 10,46. Тираж 4000 экз. Зак. № 2-332. Цена 90 к.

Издательство «Техника», 252601, Киев, 1, Крещатик, 5.

Отпечатано с матриц Головного предприятия республиканского производственного объединения «Полиграфкнига», 252057, Киев, ул. Довженко, 3 на книжной фабрике «Жовтень», 252053, Киев, Артема, 25.