КУЙБЫШЕВСКИЙ
ОРДЕНА ТРУДОВОГО
КРАСНОГО ЗНАМЕНИ
АВИАЦИОННЫЙ ИНСТИТУТ
имени С. П. КОРОЛЕВА

А. И. Данилин

ПРИКЛАДНОЕ МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ САПР. МЕТОДЫ ПРОЕКТИРОВАНИЯ И ДОКУМЕНТИРОВАНИЯ ПРОГРАММ

КУЙБЫШЕВ
1983

Министерство высшего и среднего специального образования
Р С Ф С Р

Куйбышевский ордена Трудового Красного Знамени
авиационный институт имени академика С.П.Королева

А.И.Данилин

ПРИКЛАДНОЕ МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ САПР. МЕТОДЫ ПРОЕКТИРОВАНИЯ И ДОКУМЕНТИРОВАНИЯ ПРОГРАММ

Утверждено редакционным
советом института
в качестве учебного
пособия

Куйбышев 1983

УДК 681.3

Д а н и л и н А.И. Прикладное математическое обеспечение САПР. Методы проектирования и документирования программ. – Куйбышев: КуАИ, 1983. – 52с.

В пособии рассматриваются основные методические приемы составления программ для ЭВМ, обсуждаются средства достижения их удобочитаемости и эффективности. Все рекомендации проиллюстрированы фрагментами программ на Фортране и ПЛ-1. Рассмотрена также ХИПО-технология проектирования и документирования программных комплексов.

Пособие предназначено для широкого круга студентов, выполняющих курсовые и дипломные работы с использованием ЭВМ, а также для слушателей ФПК ИТР.

Ил. 7. Библиогр. – 13 назв.

Рецензенты: к.т.н. доц. О.А. К а ц ю б а, К.А. З а й ц е в

В В Е Д Е Н И Е

Последние несколько лет являются ключевыми в развитии вычислительной техники и внедрении ее в народное хозяйство. За один 1980г. выпущено столько ЭВМ, сколько за предшествующие десять лет. Вычислительные машины и математические методы применяются сейчас во всех отраслях науки и техники при проектировании новых изделий и при их изготовлении. Созданы мощные программные комплексы, такие как системы автоматизированного проектирования (САПР) и технологической подготовки производства (АСТПП), внедрение которых приводит к тому, что все большее число инженеров втягивается во взаимодействие с ЭВМ и овладевает программированием.

В каждой области знаний имеется набор методических приемов, которыми должен владеть любой квалифицированный специалист. Не является исключением и программирование для ЭВМ. Однако, несмотря на то, что для студентов всех специальностей читаются большие курсы по языкам программирования, подкрепляемые практическими занятиями, домашними заданиями, а для некоторых специальностей и курсовыми работами, - вопросам технологии программирования практически не уделяется внимания и студенты не получают необходимых методических навыков. Данное пособие призвано восполнить этот пробел и посвящено рассмотрению основных методических приемов и правил их применения в программировании. Пособие следует изучать после усвоения материала курса "Программирование и алгоритмические языки" и решения как минимум одной задачи с применением ЭВМ.

I. ОТЛИЧИТЕЛЬНЫЕ ОСОБЕННОСТИ ХОРОШЕЙ ПРОГРАММЫ ДЛЯ ЭВМ

"Дома строятся для того, чтобы в них жить, а не смотреть на них; поэтому следует оказывать предпочтение пользе перед гармонией, за исключением тех случаев, когда можно достичь и того, и другого".

Ф. Бекон. "О правосудии".

I.1. Программа работает и легко анализируется

Все, кто решает задачи, используя при этом вычислительную технику, очень скоро устанавливают для себя, что самое важное свойство программы заключается в том, что она работает. Совершенно очевидно, что целью программирования является не создание программы, а получение результатов вычисления. Однако в некоторых случаях имеет смысл делать различие между программой, которая работает, и программой, которая работает согласно техническому заданию. По ряду причин программист может ошибочно истолковать требования технического задания, сами требования могут оказаться неоднозначными или неточными, могут измениться за время разработки программы или программист вдруг обнаружит, что не может выполнить все требования и реализует лишь часть из них.

Методы создания работающих программ, а также программ, работающих согласно техническому заданию, будут рассмотрены во второй и третьей главах - здесь же остановимся кратко на методах создания программ, позволяющих легко их анализировать и понимать.

Программы и программные комплексы неизменно и непрерывно усложняются, в связи с этим не следует забывать одно важное обстоятельство: несмотря на то, что программы пишутся для ЭВМ, читаются они, в первую очередь, людьми. Это вызвано необходимостью корректировки, использования и модификации программы. Кроме того, прог-

рамма - это документ для последующего использования, учебный материал по кодированию алгоритма и средство дальнейшей разработки более совершенных программ. Поэтому программист всегда должен следить за удобочитаемостью своей программы, хотя порой это бывает сложно. Удобочитаемость облегчает анализ самых сложных программ. Трудночитаемые программы практически невозможно проанализировать и понять, а значит и модифицировать для своих нужд, особенно если это приходится делать не автору программы. Как правило, легче полностью переписать чужую программу, чем ее модифицировать. Однако, когда программист обращается к чужой программе и видит, что она хорошо организована, легкочитаема и понятна, то естественные отрицательные эмоции, связанные с необходимостью дешифровки чужих замыслов, исчезают. Легкочитаемая программа ясно и недвусмысленно показывает, что ее автор хорошо знал, что делал, поэтому программа должна передавать логику и структуру алгоритма, а также его функциональное назначение настолько доступно, насколько это возможно.

Рассмотрим некоторые средства, позволяющие при правильном их использовании получать такие удобочитаемые, а следовательно, и легко анализируемые программы.

I.I.I. Комментарии. Желательность комментариев, казалось бы, очевидна, однако далеко не всегда их включают в программу. Некомментированная программа - это наихудшая из ошибок, которую может сделать программист, а также свидетельство дилетантского подхода и основание для понижения оклада. Подробный комментарий является хорошей предпосылкой быстрой и легкой отладки программы, поэтому можно рекомендовать следующее: во-первых, не менее четырех-пяти строк комментария на каждый модуль и в среднем по одному комментарию на две-три строки исходной программы. Комментарии следует включать в процессе написания программы - тогда в наибольшей степени вы знаете все детали и нюансы программы. Редко удается получить удовлетворительные результаты при более поздней вставке комментариев, поскольку приходится вспоминать заново, что нужно прокомментировать.

Во вторых, хорошие комментарии написать непросто. Поскольку назначение комментариев - облегчить понимание программы, то они должны быть так же хорошо продуманы, как и кодировка программы. При этом важно помнить, что комментарий должен пояснять не ЧТО программа делает (это ясно из текста программы), а ЗАЧЕМ и ПОЧЕМУ.

Если вы испытываете трудности при составлении комментария, то, скорее всего,"вы не ведаете, что творите" и вам следует еще раз обратиться к блок-схеме и функциональному описанию алгоритма.

Существует три вида комментариев: вводные, оглавления и пояснительные. Каждая программа или подпрограмма должна начинаться с вводного комментария, поясняющего, для чего она предназначена. Минимальная информация во вводном комментарии должна включать:

назначение программы;

список и функциональное содержание всех основных переменных и массивов;

указания по вводу-выводу. Список и функциональное назначение всех файлов;

список используемых подпрограмм и их назначение;

название применяемых математических методов и ссылки на литературу, откуда взят алгоритм;

сведения о примерном количестве выполняемых операций, которые нужны для оценки быстродействия программы.

сведения об авторе программы и дата ее создания.

Все эти данные необходимы для документирования программы, и лучшим местом для их размещения является сама программа. Если программа очень большая или в ней вызывается много подпрограмм, то целесообразно в ее начале помещать оглавление в виде комментария. Оглавление содержит название, функциональное назначение каждого программного модуля и условия его вызова. Пояснительные комментарии нужно давать всегда, когда кодируется один или группа операторов, выполняющих единственную функцию. Весьма существенно содержание комментария, и нет необходимости переводить с языка программирования каждый оператор. Комментарии должны объяснять цель группы операторов, а не описывать действия, производимые этими операторами.

I.I.2. Мнемоничность имен переменных. Имена переменных выбирают так, чтобы наилучшим образом определять те величины, которые они представляют. Правильный выбор имен переменных - залог удобочитаемости программы. Если ограничения на размер имени отсутствуют, то используются имена настолько длинные, насколько это необходимо для обозначения функционального наполнения каждой переменной.

Например, в операторе

$$X = Y + Z$$

имена переменных выбраны неудачно, поскольку совсем не использована мнемоника. Такая запись оператора

```
NOVOE_SOSTOYANIE=KOORDINATA+PEREMESHENIE;
```

намного лучше.

Соответствующая мнемоника должна быть использована и при выборе имен для программ, процедур, функций и подпрограмм. Программные метки должны соответствовать меткам, которые использовались в блок-схемах или в функциональных схемах, или при анализе задачи, что на практике часто не выполняется. Очень часто в программах можно увидеть переменные типа `ORK27`, `WSMQ` и т.д., расшифровать которые не всегда удается даже автору, если он некоторое время не видел свою программу.

I.I.3. Пропуск строк и пробелы. П р о п у с к с т р о к - часто недооцениваемый способ улучшения наглядности программы. Он требует лишь вложения пустой перфокарты в исходную колоду или ввода пустой строки с терминала и используется для вертикальной разрядки текста. Как в естественном языке мы пользуемся пропуском строк для отделения параграфов, так и в программе посредством этого способа можно выделять ее отдельные фрагменты. Пропуском одной строки (одна чистая перфокарта) нужно отделять каждую группу логически связанных операторов, пропуском двух строк - логически законченный фрагмент программы. Пропуск строки должен также следовать за каждой командой или оператором передачи управления, указывая на нарушение последовательности выполнения команд.

П р о б е л ы следует ставить везде, где это приведет к облегчению читаемости программы. В изъятии пробелов из программы не больше смысла, чем в том, чтобы убрать их из текста. Попробуйте прочитать, например: Какхорошописатьбезпробеловполучаетсяоченьбыстро.

Можно написать оператор Фортрана

```
DO105I=1,25,2,
```

однако значительно лучше такая запись:

```
DO  105   I = 1,25,2.
```

Использование пробелов в сочетании с пропуском строк является одним из основных способов достижения удобочитаемости.

Рассмотренные приемы повышения удобочитаемости программ являются ключевыми, и их использование необходимо определять уже в

техническом задании на программу. Суть других приемов, применение которых желательно, заключается в следующем:

1. В объявлениях следует располагать имена переменных по алфавиту, что облегчит их поиск при анализе работы программы.

2. При выборе имен записей нужно использовать имена, ориентированные на запись, а не на ее обработку. Использование одних и тех же имен для одинаковых файлов в различных программах ведет к быстрой идентификации файла.

3. Если оператор переносится на следующую строку, то перенос необходимо делать после знака операции.

4. При размещении операторов помните - одного оператора в строке достаточно.

5. Следует использовать скобки чаще, чем это необходимо. Для примера сравните выражения, выполняющие одинаковое действие:

$A*B/C*D/E*F \quad \sim (A*B*D*F)/(C*E);$

$A**B*C \quad \sim (A**B)*C;$

$A/B/C/D \quad \sim ((A/B)/C)/D.$

Форма записи в правом столбце лучше поясняет суть дела.

В заключение отметим, что небольшие усилия, которые необходимо приложить для того, чтобы сделать программу удобочитаемой, обходятся значительно дешевле, чем издержки по пересмотру, обнаружению ошибок или переделке плохо написанной программы.

Способность писать удобочитаемые программы - отличительная особенность хорошего программиста. В оправдание плохо составленных программ обычно приводят аргумент - это сделанная на скорую руку черновая программа ограниченного использования. Но программы имеют тенденцию жить дольше и использоваться шире, чем запланировано, поэтому так важно учиться писать сразу хорошую программу. Кроме того, это значительно сэкономит время при тестировании и модификации программы.

I.2. Минимальные затраты на сопровождение программы

Всякая программа, которая хоть чего-нибудь стоит, будет находиться в обращении в течение долгого времени. Независимо от того, будете ли вы, как первоначальный автор программы, обеспечивать ее эксплуатацию (сопровождать программу) или передадите ее кому-ни-

будь другому, существенно то, что почти всегда последующие дополнения и сопровождение программы необходимы.

Существует несколько аспектов проблемы сопровождения; некоторые из них могут быть решены путем правильного проектирования программ, другие - правильного составления технической документации на программы. Основные трудности, с которыми сталкиваются при сопровождении, состоят в следующем:

I. Программы, переданные в эксплуатацию, все еще содержат значительное число ошибок; таким образом, то, что называют сопровождением, в действительности является продолжением этапа испытаний. Факт кажется очевидным, но часто игнорируется программистами-разработчиками.

2. Модернизация программ при введении новых компиляторов, операционных систем и других систем математического обеспечения; при включении программы в комплекс программ; при подключении к комплексу программ новых информационных подсистем и т.д. Маловероятно, что эта проблема будет решена в ближайшем будущем.

3. Большинство важнейших программ требует постоянного сопровождения в связи с изменениями в составе и потребностях пользователя. Практически не существует прикладных программ, которые так хорошо определены, что не требуют каких-либо изменений. При этом часто оказывается, что программиста-разработчика невозможно найти.

4. Большинство людей испытывают затруднения в понимании чужих программ по той причине, что многие программисты пишут свои программы довольно неорганизованно и выдают трудночитаемые программы.

5. Описания, сопровождающие большинство программ, а также систем программного обеспечения - ужасны. Как правило, описания программ и инструкции по эксплуатации создают программисты-разработчики, многие из которых считают этот этап работы неизбежным злом. Нам неизвестны программисты, которым бы нравилось оформлять свои программы. Ясно, что многие организации сейчас расплачиваются за низкие требования к оформлению программ в прошлом. Сейчас появились способы ведения документации, благодаря которым создание документации является логическим продолжением этапа проектирования программ, включено в сам процесс создания программы и не требует от программиста дополнительных усилий - все это позволяет предположить, что в скором будущем проблема будет решена.

Правильное проектирование программ, очевидно, не может разрешить всех проблем сопровождения. Однако, создавая программу, мы

должны постоянно иметь в виду, что ее сопровождение почти наверняка будет выполняться кем-то другим. И если программа проста в отладке и тестировании, удобочитаема, то и ее сопровождение будет относительно легким как для программиста-разработчика, так и для остальных. Для достижения легкости отладки важно:

1. Использовать структурное программирование или одну из форм модульного программирования; избегать хаотического построения программ. Не ясно, сможет ли по истечении некоторого времени понять такую программу программист-разработчик, но совершенно ясно, что большинство программистов, обеспечивающих сопровождение - не смогут.

2. Избегать сложности стиля в программировании. Создавая программу, помните, что трюки в программировании вызывают лишь раздражение.

К примеру, можно довольно изящно выполнить формирование единичной матрицы с помощью операторов Фортрана

```
      DO   10 = I, N
      DO   10 = J, N
10    A(I, J) =(I/J)*(J/I)
```

Однако от другого программиста, читающего такую программу, анализ этого фрагмента потребует дополнительных усилий и вызовет лишь отрицательные эмоции. Значительно проще было бы написать

```
DO  10  I = I, N
DO  10  J = 1, N
 A(I,J) = 0.
IF(I.EQ.J) A(I,J)= 1.
10  CONTINUE ,
```

тем более, что по количеству выполняемых операций вторая запись экономичнее и программа будет работать быстрее.

3. Оформлять свои программы как для самого себя.

Разумеется, если бы все программисты следовали этим указаниям, у нас было бы очень мало трудностей в сопровождении. По-видимому, до тех пор, пока в программе имеются ошибки, с ней должны работать те программисты, которые знают ее свойства, вместо того, чтобы перекладывать эту работу на людей, которые не понимают программу и не хотят ее знать. Поэтому, прежде чем принять программу к сопровождению, следует провести некоторую проверку того, а испытана и оформлена ли она в соответствии с разумными нормами программирования

и прошла ли опытную эксплуатацию у пользователя (под надзором разработчиков или без него).

I.3. Эффективность программы

Большинство программистов считают это свойство наиболее важной характеристикой программы и готовы тратить часы и даже дни на увеличение ее быстродействия и уменьшение потребной памяти. Однако затрачиваемые усилия не оправдывают себя, если не решается при этом основная задача программирования - получение правильной программы.

> Программы создаются для решения задач, и эффективная программа не нужна, если она не обеспечивает правильных результатов (правило Ван Тассела).

Таким образом, правильность программ первична, а ее эффективность - вторична. Эффективную, но неправильную программу редко можно сделать правильной, в то время как правильную, но неэффективную программу можно оптимизировать и сделать эффективной - поэтому оптимизация является вторым этапом программирования.

Многие методы, делающие программу эффективной, не наносят ущерба ее удобочитаемости, их следует использовать всегда. Однако существуют приемы, которые просто вредны для получения удобочитаемой программы, а удобочитаемость программы более существенна, чем ее эффективность, поскольку удобочитаемую программу легче отлаживать, модифицировать и использовать. Лишь в особых случаях программу следует делать более эффективной в ущерб удобочитаемости: программа либо не помещается в памяти, либо слишком долго выполняется, либо должна быть включена в библиотеку и часто использоваться.

Прежде чем начать оптимизировать программу, необходимо тщательнейшим образом проверить алгоритм с точки зрения потребного для решения задачи объема памяти и количества операций. Возможно, что вы ошиблись в самом начале и приняли неудачный алгоритм - в этом случае никакая оптимизация не позволит вам добиться принципиально достижимых результатов, и лучшей оптимизацией явится полное перепрограммирование задачи. (Некоторые аспекты проблемы выбора алгоритма рассматриваются во втором разделе).

На следующем этапе нужно проанализировать время работы всех частей (подпрограмм) вашей программы. Большинство программ имеет одну критическую область, которая использует большую часть времени выполнения. Как установлено Д.Е. Кнутом при исследовании Фортран-программ [7], около 50% времени центрального процессора расходуется на выполнение 5% операторов исходной программы . Отсюда явствует, что оптимизации следует подвергать только критические области программы, оптимизация же некритических частей программы - пустая трата времени.

Эффективность программы в процессе выполнения определяется использованием двух ресурсов: потребным для работы временем и памятью, необходимой программе. Для программиста более важный фактор - время, так как в большинстве случаев программа оценивается количеством машинного времени, необходимым для ее выполнения. За рубежом на вычислительных центрах коллективного пользования введена новая система оплаты - за объем используемой памяти, поскольку традиционный критерий оплаты - машинное время - здесь неприменим. В нашей стране пока не получили широкого распространения ВЦ коллективного пользования, но, видимо, в скором будущем и нам придется решать сходные проблемы, поэтому вопросы экономии памяти при кажущейся их второстепенности могут оказаться решающими, особенно для организаций, не имеющих собственной вычислительной техники.

Кроме того, в настоящее время прочно входят в нашу повседневную жизнь дешевые, надежные и неприхотливые в обслуживании мини-ЭВМ, для которых вопросы эффективного использования памяти имеют первостепенное значение в силу ее небольшого объема. И, наконец, о пользе экономии памяти свидетельствует и чисто психологический фактор: с ростом объема располагаемой памяти объем программируемых задач увеличивается таким образом, чтобы заполнить всю имеющуюся память. Однако экономное использование памяти почти всегда сопровождается увеличением времени программирования и отладки, а также увеличением времени выполнения программы; поэтому, если память не используется полностью, ее распределением не интересуются до тех пор, пока памяти достаточно.

Рассмотрим ряд приемов, с помощью которых можно повысить эффективность программ. Сначала обратимся к способам экономии памяти, а затем отметим приемы повышения быстродействия программ.

I.3.I. Оверлейность программы. Под оверлейностью программы понимают возможность последовательного вызова различных частей

программы - сегментов - на одно и то же место оперативной памяти для их выполнения. Оверлейность, пожалуй, единственный способ выполнения больших по объему программ. Однако, чтобы пересылать сегменты из периферийной памяти (обычно с магнитного диска) в оперативную, необходимо дополнительное время.

Если программу можно сделать оверлейной, то следует продумать размещение данных. Каждый раз, когда сегмент вызывается с внешнего носителя, он считается неизменным. Данные, полученные в результате работы такого сегмента, если не принять специальные меры, теряются для дальнейшей обработки.

Каждый оверлейный сегмент программы должен всю необходимую информацию получать от корневой, управляющей программы и ей же передавать результаты своей работы, поэтому организация оверлейности программы требует дополнительных расходов времени высококвалифицированных программистов и машинного времени.

I.3.2. Виртуальная память. Возможность оверлейности реализуется также при использовании виртуальной памяти. Если машина имеет виртуальную память, то операционная система автоматически делит программу на части фиксированной длины, называемые страницами. Операционная система в случае необходимости пересылает страницы в оперативную память или из памяти на внешний носитель. Таким образом, организация оверлейности перестает быть обязанностью программиста и возлагается на машину. Но это, в свою очередь, ведет к неоправданно большим расходам машинного времени при выполнении программы. К примеру, если по ходу решения задачи необходимо работать с массивом или частью программы, которой нет в оперативной памяти, то теряется время на их считывание с внешнего носителя. Классической иллюстрацией неэффективного выполнения программы на машине с виртуальной памятью является следующий фрагмент Фортран-программы:

```
      DO  15  K=1,  5ФФ
          DO  15    L=1,2Ф
15        X(K,L)=Ф.
```

Массивы в Фортране хранятся по столбцам, и каждый столбец занимает одну страницу. Таким образом, при выполнении оператора с меткой I5 нужно каждый раз из I0.000 обращаться к другой странице. Решение проблемы состоит в том, чтобы сделать цикл по K =1,500 внутренним, в этом случае программа присвоит нулевые значения 500

элементам массива X прежде, чем понадобится новая страница. Так просто изменив последовательность циклов, мы сможем уменьшить количество обращений к внешнему носителю в 500 раз.

Из приведенного примера вытекает, что виртуальная память, снимая с программиста заботы по распределению и управлению памятью, добавляет ему новые, связанные с анализом работы своей программы в свете страничной организации обрабатываемых данных. Зарубежный опыт работы с виртуальной памятью [5] показывает, что на таких машинах с виртуальной памятью практически невозможно добиться высоких результатов по быстродействию.

I.3.3. Заголовки сообщений. Заголовок может состоять из нескольких строк, причем в некоторых строках содержится повторяющаяся информация. В этом случае следует не определять всю строку как символьную константу (литерал), а использовать в операторах ввода-вывода коэффициенты повторения. Подпрограммы формирования заголовков, поскольку они обычно расходуют много памяти и редко используются, следует помещать в оверлейные сегменты.

I.3.4. Эквивалентность. Если одна переменная используется в одной части программы, а вторая переменная нужна в другой части, то обе эти переменные могут занимать одно и то же место оперативной памяти, так как они используются в программе в разное время. Соответствующие операторы установления эквивалентности переменных имеются практически во всех языках программирования. Однако при их использовании для сохранения удобочитаемости эквивалентность переменных должна быть соответствующим образом прокомментирована.

I.3.5. Вычисление констант. Очень часто в программах можно встретить выражения, подобные

$Y = COS(3.14159/3.)$

$T = X - 32 \phi\phi./12.$ и т.д.,

которые содержат константы. К программированию таких выражений следует подходить с большой осмотрительностью. Дело в том, что некоторые компиляторы вычисляют значения констант во время компилирования, а некоторые - во время выполнения программы. Причем узнать как работает конкретный компилятор, обычно невозможно, так как этот аспект в техническом описании, как правило, не освещается.

Если константы вычисляются во время выполнения программы, то ясно, что нас ждет непроизводительный расход машинного времени и памяти.

Поэтому возьмите себе за правило: все константы вычислять на бумаге, а не на ЭВМ.

I.3.6. Арифметические операции. Арифметические операции выполняются с различной скоростью. Перечислим операции в порядке увеличения времени их выполнения: сложение и вычитание; умножение; деление; возведение в степень.

В некоторых случаях медленно выполняемые операции легко заменить на более быстрые. Сложение выполняется быстрее, чем умножение, поэтому умножение на небольшое число должно быть заменено сложением. Так, $3*J$ нужно заменить на $J+J+J$. Однако нужно проявлять чувство меры и всегда следить за тем, чтобы суммарное время всех операций сложения не превысило времени выполнения одной операции умножения. Кроме того, для чисел с плавающей точкой замена умножения сложением может привести к потере точности, так как ошибки округления при сложении имеют тенденцию накапливаться.

Умножение выполняется, по меньшей мере, в два раза быстрее деления, поэтому всюду, где возможно, деление следует заменять умножением: вместо A/4 пишите A * 0.25.

Будьте внимательны при программировании формул: возможно, есть смысл преобразовать выражение. Например, $X=2.*B+(A-1.)/D+2.*R$ можно заменить на $X=2.*(B+R)+(A-1.)/D$, что исключает одну операцию умножения.

Если в вычислениях приходится все время делить на некоторое число, например на X, то следует заменить его на обратную величину - и тогда операция деления преобразуется в умножение.

Важно также правильно задать тип показателя степени в операции возведения в степень. Если показатель степени - целое число, то возведение выполняется последовательным умножением. Если же показатель-число с плавающей точкой, то результат вычисляется через логарифм. Но в том и в другом случае для выполнения операции возведения в степень вызывается библиотечная программа, которая расходует и время, и память, поэтому заменяйте:

1) $X**2$ на $X*X$;

2) $X**3$ на $X*X*X$;

3) $X**4$ на $AX=X**X$; $AX*AX$;

4) $B=A**P$ на $JP=P$; $B=A**JP$;

если P - целое число в форме с плавающей точкой.

Функция извлечения квадратного корня обычно реализуется гораздо быстрее, и точность при этом выше, чем при выполнении операции возведения в степень, поэтому вместо `A**.5` всегда пишите `SQRT(A)`.

I.3.7. Смешанные типы данных. Смешанные типы данных получаются в результате использования в арифметических и логических выражениях переменных и констант различного типа. Если вы смешиваете типы данных, то преобразования обязательны, что, в свою очередь, влечет за собой размещение в памяти и выполнение библиотечных программ, обеспечивающих эти преобразования. По этой причине следует всячески избегать смешения типов данных, в первую очередь это относится к записи констант, поскольку именно здесь программисты зачастую допускают небрежность. Дело в том, что многие компиляторы преобразуют константы в нужный тип во время выполнения программы, попусту расходуя память и время, в то время как избежать этого исключительно просто. Например, при выполнении Фортран-программы

```
      DO 10 N=1, 1000
10    A(N)=0
```

Требуется 1000 преобразований целого нуля в форму с плавающей точкой. Замена же оператора с меткой `10` на `A(N)=0.0` исключает эти затраты. Подобная ситуация возникает при использовании операторов `Y=1/X, A=2*B, C=3*D+1,` в которых преобразования необходимы, в то время как для операторов `A=2.*B, C=3.*D+1, Y=1./X.` их не требуется.

I.3.8. Повторяющиеся вычисления. Может показаться тривиальным утверждение, что никакие операции не следует повторять, однако повторяющиеся вычисления можно найти практически в любой программе, например:

```
SIGMA 1=SIN(THETA)+SIN(THETA)**2

SIGMA 2=SIN(THETA)/3.0
```

Более эффективной будет такая программа:

```
A=SIN(THETA)
SIGMA 1=A+A*A
SIGMA 2=A*0.333
```

Здесь `SIN(THETA)` вычисляется один раз вместо трех в исходном тексте.

Еще пример: вычисление корней квадратного уравнения

```
R1 = (-B+SQRT(B**2-4.*A*C))/(2.*A)
R2 = (-B-SQRT(B**2-4.*A*C))/(2.*A)
```

Более эффективной, но менее удобочитаемой программой будет такая:

```
      D = 1./(A+A)
DIS = SQRT(B*B-4.*A*C)
 R1 = (-B+DIS)*D
 R2 = (-B-DIS)*D
```

Хотя вторая группа операторов длиннее, она выполняется быстрее и использует меньше памяти. Дешевле чем двойное вычисление `SQRT(B**2-4.*A*C)`, обходится двукратное обращение к `DIS`, так как пересылки очень быстры.

П о м н и т е! Меньшее число операторов исходного текста еще не доказывает эффективность программы по времени и памяти.

I.3.9. Организация циклов. Значительная часть времени при использовании циклов тратится на инициирование и проверку управляющей переменной цикла. Тщательная организация вложенных циклов способствует экономии времени. Для примера рассмотрим тройной цикл в программе на ПЛ-I:

```
DO I=1 TO 2Ø BY 1;        Инициирование I раз
DO J=1 TO 1Ø BY 1;        Инициирование 20 раз
DO K=1 TO  5 BY 1;        Инициирование 200 раз
операторы тела цикла      Выполнение I000 раз
   END;                   Завершение I000 раз
 END;                     Завершение 200 раз
END;                      Завершение 20 раз.
```

Таким образом, инициирование циклов выполняется 221 раз, а завершение I220 раз. Рассмотрим другую организацию тех же циклов:

```
DO K=1 TO 5 BY 1;         Инициирование  I раз
DO J=1 TO 1Ø BY 1;        Инициирование  5 раз
DO I=1 TO 2Ø BY 1;        Инициирование  50 раз
```

оператора тела цикла

```
END;
END;
END;
```

Выполнение 1000 раз
Завершение 1000 раз
Завершение 50 раз
Завершение 5 раз

Итого имеем 56 инициирований и 1055 завершений. По сравнению с предыдущим способом чистая экономия составляет 165 инициирований и 165 завершений циклов. Отсюда можно вывести о б щ е е п р а в и л о: старайтесь так организовать циклы, чтобы внешний цикл имел наименьшее число итераций.

1.3.10. Оптимизация циклов. Совершенно ясно, что оптимизации следует подвергать в первую очередь внутренние циклы. К примеру, в тройном цикле по 100 итераций каждый любая, даже самая ничтожная экономия во внутреннем цикле, будет увеличиваться в 100^3 = 1.000.000 раз и в итоге даст значительный выигрыш по машинным операциям.

Основной ошибкой, довольно часто встречающейся даже у опытных программистов, является наличие повторяющихся вычислений.

Рассмотрим фрагмент Фортран - программы:

```
      DO 15 I=1,10
      DO 15 J=1,10
15    A(I,J)=B(I,J)+D/I+D/K
```

При внимательном анализе можно заметить, что во внутреннем цикле по J выражения D/I и D/K не меняются, однако время на их вычисление расходуется. Более эффективной будет такая программа:

```
      DK=D/K
      DO 15 I=1,10
      DI=D/I+DK
      DO 15 J=1,10
15    A(I,J)=B(I,J)+DI
```

Здесь все повторяющиеся вычисления вынесены за тело цикла.

Иногда издержки по организации завершения тела цикла оказываются слишком дорогими (это можно выяснить из анализа ассемблерного листинга скомпилированной программы), в этом случае имеет смысл преобразовать цикл, например,

```
      DO   1ф   J = 1,1 ффф
1ф    A(J) = ф.
```

Представление цикла в виде

```
      DO   1ф     J = 1,1 ффф, 2
      A(J) = ф.
1ф    A(J+1) = ф.
```

сокращает операции завершения цикла в два раза. Описанный прием называется р а з в е р т к о й ц и к л а.

Если в программе имеются несколько циклов с одинаковым диапазоном и шагом изменения управляющей переменной, то, если не используется машина с виртуальной памятью, такие циклы следует объединить в один:

неэффективно

```
      DO   11   J = 1.5 фф
11    A(J) = ф.
      DO   12   I = 1,5 фф
12    B(J) = ф.
```

эффективно

```
      DO   11   J = 1,5 фф
      A(J) = ф.
11    B(J) = ф.
```

I.3.II. Индексация. Индексы полезны, хотя они расходуют и машинное время, и память. Ни один программист не захочет отказаться от использования индексов независимо от того, насколько при этом повысится эффективность. Для повышения быстродействия при использовании индексов можно предпринять некоторые действия:

I. Если внутри одного или нескольких операторов к индексированной переменной обращаются более одного раза, то следует присвоить ее значение простой переменной. Например, выражение

```
X = (A(J)+1./A(J))*C+A(J)
```

следует привести к виду:

```
AJ = A(J); X = (AJ+1./AJ)*C+AJ.
```

2. Не нужно применять индексацию там, где можно обойтись простой переменной:

неэффективно

```
      DO 12 J=1,1ФФ
      DO 12 K=1,1ФФ;
12    B(K)=B(K)+A(J)
```

эффективно

```
      DO 12 J=1,1ФФ
      AJ=A(J)
      DO 12 K=1,1ФФ
12    B(K)=B(K)+AJ
```

Индексация обладает еще одной особенностью: чем больше индексов используется, тем менее эффективна программа. Это значит, что массив A (I Ø Ø Ø) более эффективен, чем массив A (IØØ, IØ). Большинство программистов охотнее работают с массивами, имеющими два и более индекса, однако если программа, в которой использованы многомерные массивы, будет часто применяться, то лучше перейти к одномерным массивам. Известен случай, когда только путем замены двумерных массивов на одномерные повысилось быстродействие программы на Алголе для БЭСМ-6 в 3,5 раза. Для программ на Фортране и ПЛ/I для ЕС ЭВМ замена двумерных массивов на одномерные увеличивает быстродействия в ДОС ЕС в среднем в 2-2,5 раза, а в ОС ЕС - в 3 раза. Объективно применение одномерных массивов вместо многомерных приводит к ухудшению удобочитаемости программы.

Преобразование двумерной матрицы $N \times M$ в вектор можно осуществить посредством следующих способов:

I. Если матрица хранится по строкам, т.е. правый индекс изменяется быстрее левого, то K -й элемент вектора вычисляется как

$$VECTOR(K)=MATRIX(I,J);\ K=M*(I-1)+J.$$

2. Если матрица хранится по столбцам - левый индекс изменяется быстрее, то

$$VECTOR(K)=MATRIX(I,J);\ K=N*(J-1)+I.$$

Для самостоятельного анализа ухудшения удобочитаемости программы приведем две программы умножения матриц. Обычная программа на языке ПЛ-I

```
      DO I=1 TO N;
      DO J=1 TO M;
       A(I,J)=ФЕФ;
      DO K=1 TO L;
```

```
      A(I,J)=A(I,J)+B(J,K)*C(K,J);
      END;
   END;
END;
```

Самая эффективная программа будет выглядеть следующим образом:

```
      DO  I=1  TO  N ;
      DO  J=1  TO  M :
      TEMP = ØEØ ;
      DO  K=1  TO  L ;
      JK = L*(J-1)+K ;
      KJ = M*(K-1)+J ;
      TEMP = TEMP+B(JK)*C(KJ);
      END ;
   IJ = M*(I-1)+J ;
   A(IJ) = TEMP ;
   END ,
END;
```

Здесь, увеличивая быстродействие, мы дополнительно расходуем память на хранение четырех новых переменных *TEMP*, *JK* , *KJ* и *IJ* .

К числу отрицательных особенностей индексации можно отнести и то, что вычисления со сложными индексами выполняются медленнее по сравнению с тем, когда индекс может быть вычислен заранее. Так запись *A(3*J+K)* менее эффективна, чем *I = 3*J+K ; A(I)* . Отсюда вывод: следует избегать вычислений со сложной индексацией.

Каждый язык программирования имеет определенный тип данных, наиболее предпочтительный для индексации. В Фортране и Алголе это *INTEGER*, в ПЛ-I- *BINARY FIXED* . Такой тип данных следует использовать всегда с целью исключить преобразования и тем самым сэкономить время. Попробуйте сами выполнить программу дважды с разными типами данных для индексов и вы наверняка получите заметную разницу во времени.

I.3.I2. Ввод - вывод. По сравнению с быстродействием процессора скорость выполнения операций ввода-вывода на несколько порядков меньше, поскольку они почти всегда связаны с механическим перемещением или носителя информации, или устройств считывания-записи, поэтому все операции ввода-вывода ВСЕГДА следует сокращать до минимума. Другими словами: не следует вводит данные, которые можно вычислить в программе, кроме того, нужно удостовериться, что каждый оператор ввода-вывода передает минимальное и только действительно необходимое количество информации.

Пожалуй, единственным способом сокращения операций ввода-вывода является считывание или запись информации большими порциями с последующей выборкой нужных данных, но в этом случае необходимо очень тщательно проектировать алгоритм обработки и определять содержание информации на внешнем носителе для достижения их гармонии, т.е. следующая порция данных должна требоваться лишь после того, как предыдущая порция будет использована полностью без остатка. Это требует большого труда и изобретательности, но эффективность программы вознаграждает затраченные усилия сторицей.

В заключение отметим, что эффективные программы позволяют нам решать такие задачи, которые в недалеком прошлом казались недостижимой мечтой. Однако не следует придавать эффективности особое значение. По мнению В.А.Вульфа [6], "во имя эффективности в программировании было совершено больше прегрешений, чем по какой-либо другой причине, включая непроходимую глупость". Поэтому правильный выбор алгоритма, рациональная организация программы и корректное программирование в большей степени определяют успех в решении задачи.

Не старайтесь писать сразу эффективную программу - сначала добейтесь ее правильности.

"Так как все в мире само по себе меняется к худшему, то, если не изменить это к лучшему силой нашего ума, где же будет предел несчастьям?"

Ф.Бэкон. "О достоинстве и приумножении наук", Кн.6, Гл.Ш.

2.I. П о с т а н о в к а з а д а ч и

Обычно разработка прикладного программного обеспечения начинается с неясного и абсолютно невинного определения потребности: "необходимо, чтобы программа вычисляла напряжения в элементах конструкции", "напишите программу, рассчитывающую параметры самолета", "нужно автоматизировать процесс проектирования узла" - все это плохо определенные задачи. Задачи, подлежащие программированию, обычно ставятся и формулируются не теми, кто будет программировать, поэтому важно, чтобы между заказчиком и разработчиком с самого начала было установлено полное взаимопонимание относительно целей и конечных результатов программирования, чего на практике добиться чрезвычайно трудно. Существует даже определение программы, пронизанное мрачной иронией относительно этого этапа работы, а именно: идеальная программа - это программа, которая дает точный желаемый выход при неформулированных требованиях пользователя. По этой причине следует как можно точнее и как можно раньше ответить на вопрос: "Чего мы хотим?" или "Чего хотят от нас?". Четко сформулированная задача - залог получения нужной пользователю программы.

Заказчик должен подготовить описание задачи, это необходимо даже в том случае, когда программист является одновременно и постановщиком задачи. Такое описание заставит заказчика серьезно обдумать задачу и четко определить ее постановку. На этом этапе работы заказчик не должен ориентироваться ни на какую ЭВМ, а четко определяет желаемый выход программы или системы, после чего должно состояться обсуждение задачи программистом и заказчиком. Такое обсуждение очень часто помогает заказчику разобраться в своих представлениях задачи. Далее программист переписывает постановку задачи, ориентируясь на конкретную вычислительную ма-

шину. Необходимо сжатое, но полное описание задачи, при этом особое внимание нужно обратить на содержательную часть входной и выходной информации (спецификации задачи) и функциональные аспекты ее обработки, иными словами, на этом этапе работы примерно намечается укрупненный поэтапный алгоритм решения, кроме того, программист определяет ограничения на размер задачи и (или) на типы обрабатываемых данных. Выбор ограничений является ключевым вопросом при постановке задачи, тесно связан с установлением целей программирования и всегда затрудняет общение программиста и заказчика. Заказчик, как правило, хорошо осведомлен о достижениях в той области знаний, из которой берется задача, он знаком с рядом программ, решающих сходные задачи, но обычно не утруждает себя рассмотрением и анализом конкретных особенностей конкретных вычислительных машин, на которых работают эти программы. Стремление получить самую универсальную, самую мощную и самую быстродействующую программу в 90% случаев приводит к тому, что проект остается нереализованным вовсе, а в 10% случаев проект реализуется настолько в ущербном виде, что даже у заказчика возникает сильнейшее сомнение относительно полезности затраченных усилий. Поэтому в самом начале проектирования устанавливайте для себя скромные цели.

Природа научно-технических задач такова, что обычно мощность прикладного программного обеспечения можно увеличить посредством механического увеличения границ массивов или количества записей во внешней памяти - это позволит, несмотря на начальные ограничения проекта, модифицировать программу для решения задач большего размера или большей сложности. Однако в этом случае при программировании необходимо иметь в виду неизбежность дальнейшей модификации и писать универсальные программы, т.е. программы, не зависящие от конкретного набора данных. Так например, программа ввода и суммирования элементов вектора может быть написана различными способами:

первый способ

```
      READ (5,15)(A(I), I=1,5Ø)
      SUM = Ø.
      DO 2Ø  I=1,5Ø
2Ø    SUM = SUM+A(I)
```

второй способ

```
      READ (5,15)(N,A(I),I=1,N)
      SUM = Ø.
      DO 2Ø  I=1,N
2Ø    SUM = SUM+A(I)
```

Первый способ не универсален, поскольку всегда требует чтения 50-ти элементов вектора. Во втором способе количество элементов определяется вводимой информацией и программа получается более универсальной, однако при этом программа уязвима для ошибок: требуется дополнительный анализ числа N на случай $N \leqslant 0$.

Относительно создания универсальных программ можно рекомендовать одно о б щ е е п р а в и л о: используйте в качестве параметров переменные, а не константы!

После определения программистом постановки задачи написанные спецификации должны быть тщательно изучены заказчиком совместно с программистом для того, чтобы была уверенность в их правильном понимании. Возможно, обсуждение придется повторить несколько раз, при этом главное, чтобы заказчик твердо уяснил, что он собирается получить именно то, что описано.

П о м н и т е! Любая небрежность, допущенная на стадии определения задачи, вызовет впоследствии осложнения и задержит завершение проекта.

После того, как задача определена, необходимо заморозить спецификации и в дальнейшем не вносить в них никакие изменения и дополнения. Если же заказчик настаивает на внесении изменений, следует сразу согласовывать увеличение сроков программирования. Многие проекты невозможно планировать из-за постоянно меняющихся спецификаций. Заказчики, считающие, что внести незначительные изменения в выходные результаты ничего не стоит, не поймут также и то, почему нельзя сделать небольшое изменение во входных данных, не вызвав при этом дополнительных расходов как машинного времени, так и времени программистов, поэтому надо приучать заказчиков дорого платить за все изменения - это невольно заставит их более тщательно продумывать и более ответственно ставить задачу.

2.2. Н и с х о д я щ е е п р о е к т и р о в а н и е п р о г р а м м

После постановки задачи начинается проектирование программы. Весь опыт создания программного обеспечения показывает, что наилучшим, как и с точки зрения временных затрат, является способ проектирования с в е р х у _ в н и з. Проектирование программ сверху вниз подобно написанию школьного сочинения. Сначала пишется

название темы, затем дается укрупненный план сочинения, т.е. сочинение разбивается на части. При написании каждой отдельной части предварительно составляется ее подробный план. При кажущемся обилии дополнительной работы такой способ является наиболее целесообразным и используется как в литературе, так и в программировании. Например: первоначальный вариант романа "Анна Каренина" занимал одну (!!) страницу рукописного текста; аналогичная ситуация возникла и при разработке информационно-поисковой системы банка данных газеты Нью-Йорк Таймс, составившей в итоге 83324 оператора исходного текста [I].

Используя метод проектирования сверху вниз, разработку проекта обычно начинают с постановки целей и определения основных задач, ведущих к достижению этих целей. Причем, вначале необходимо написать то, что вы хотите сделать, на естественном языке. Если вы не в состоянии это сделать, нельзя надеяться, что удастся составить программу.

Метод проектирования сверху вниз предполагает сначала определение задачи в общих чертах, а затем постепенное ее уточнение путем внесения более мелких деталей. На каждом шаге необходимо выявлять основные функции, которые нужно выполнить, т.е. данная задача разбивается на ряд подзадач, те, в свою очередь, еще на ряд подзадач и так до тех пор, пока эти подзадачи не станут настолько простыми, что каждой из них будет соответствовать один программный модуль. Действие каждого модуля должно быть описано одной фразой. Если описание модуля занимает больше одной строки - детализация продолжается. Далее следует описать данные, указывая их структуру и основные процессы обработки. В описание включают тщательно отобранные примеры, убедительно демонстрирующие ФУНКЦИИ системы и их наиболее существенные варианты. Такие примеры будут полезны позже на стадии тестирования системы, которое неизбежно.

Проектирование должно быть завершено до начала программирования. Следует сделать несколько итераций проекта и попытаться "выполнить" будущую программу, сидя за столом, что при правильном и полном проекте, разбитом на уровни иерархии, вполне возможно, поскольку определены все функции модулей и их логическая взаимосвязь. Если этого сделать не удается, то проект дорабатывается. К сожалению, при создании небольших прикладных программ этап проектирования опускается, и программисты сразу пишут программы в

в форме кода, возможно это и экономит время, но если в дальнейшем такая программа будет использоваться другими людьми, то отсутствие этапа проектирования скажется на значительном увеличении затрат на сопровождение. Дело в том, что одним из основных результатов проектирования программы является техническая документация на программу. Соотнесение исходного текста программы с проектной документацией позволяет быстрее и лучше понять программу и более эффективно ее эксплуатировать.

2.3. Модульное программирование

Логическим продолжением нисходящего проектирования программ является модульное программирование, иначе - представление всей программы в виде отдельных фрагментов и последовательное программирование каждого фрагмента (модуля). Если задача проектируется сверху вниз, то она, естественно, разбивается на подзадачи для возможных модулей. При этом преследуются две цели:

чтобы программный модуль был правильным и не зависел от контекста, в котором он будет использован;

чтобы из определяемых модулей можно было формировать большие программы без каких-либо предварительных знаний о внутренней работе и строении модуля.

2.3.1. Определение модуля. Монофункциональность. Модуль реализует одну функцию и имеет один вход и один выход. Каждый модуль имеет свое назначение, отличающееся от назначения других модулей, представляет собой замкнутый блок, вход и выход которого точно определены. Кроме того, вся порождаемая модулем информация должна быть точно определена и полностью специфицирована.

Независимость. Требуется, чтобы модуль не зависел от: источника входных данных; места назначения выходных данных; предыстории процесса. При разбиении программы на функциональные блоки можно добиться некоторой самостоятельности модулей, хотя определенная зависимость между ними все-таки существует; например, работа модуля А зависит от параметра, который вычисляется в модуле В.

Р а з м е р м о д у л я. Многие специалисты в области программного обеспечения считают эту характеристику модуля важнейшей. Однако не следует забывать тот очевидный факт, что если фрагмент программы, пусть даже небольшого размера, не реализует единственную функцию, имеет несколько входов и выходов и не обладает относительной независимостью, то назвать этот фрагмент модулем нельзя. С другой стороны, сильнейшее сомнение вызывают модули длиной в 200 и более операторов исходного текста. Автор будет очень благодарен тому, кто сообщит ему о функции, не допускающей дробления на логически законченные подфункции, которая требует для своего определения так много операторов.

Для размера модуля рекомендуем необязательное правило: модуль должен помещаться на одной странице листинга.

2.3.2. Методы достижения модульности. Хорошие модули подобны математическим функциям. Для примера рассмотрим функцию квадратного корня, программируемую как $SQRT(X)$. Функция $SQRT$ может быть вычислена только для неотрицательных чисел $X \geq 0$, поэтому областью определения функции являются неотрицательные числа. Область определения для модуля - это набор всех возможных значений входных данных. Аналогично набор возможных выходных данных является областью значений. Модуль должен всегда проверять соответствие входных данных области определения, а выходных - области значений - это исключит неправильное использование модуля. Если же входные или выходные данные не соответствуют своим областям, то модуль должен выдавать диагностическое сообщение. Такая диагностика называется п о б о ч н ы м э ф ф е к т о м. Заметим, что побочные эффекты у модулей также должны планироваться и проектироваться. На практике же этого, как правило, не делается, и очень часто можно наблюдать, как машинный сбой вызывает повторение трехчасовой программы, в то время как при соответствующей диагностике и организации вычислений можно было бы вернуться назад не на три часа, а на десять минут.

Таким образом, если для модуля точно определены выполняемая функция; область допустимых входных значений; область возможных выходных значений; побочные эффекты, то результатом явятся четко составленные модули, а следовательно и хорошие программы.

Дадим ряд практических рекомендаций, которым нужно следовать при написании модулей:

I. Используйте символьные параметры. Многие программисты зачастую действуют, исходя из предположения, что им никогда не придется изменять программу - в результате многие параметры программы кодируются в виде констант. Таким образом, при изменении требований к программе приходится изменять КАЖДЫЙ оператор исходного текста, где имеется такая устаревшая константа. Чтобы избежать трудностей, связанных с изменением устаревших констант, нужно стремиться задать как можно больше параметров в символьной форме и присваивать конкретные значения таким переменным. Это позволит изменять параметры программы изменением только ОДНОГО оператора.

При кодировании модуля мысленно задавайте вопросы: "Что нужно изменить, если массив *SMATR* станет больше?"; "Какие переделки потребуются, если количество записей в файле *STRESS* станет больше?"; "Что изменится, если изменится формат входных данных?" и т.д.

2. Отделяйте действия по вводу-выводу от вычислительных операций. Это очень важное правило: многие программы оказываются немодульными только потому, что операции ввода-вывода в них рассеяны по всей программе. Техническая база ЭВМ постоянно улучшается, и если сегодня вашей программе требуется ввод с перфокарт, то вполне вероятно, что завтра эти же данные будут вводиться с удаленного терминала или же порождаться какими-либо другими, психологически более удобными устройствами ввода. Поэтому все действия по вводу-выводу нужно вынести в отдельные модули, не смущаясь тем, что в модуле содержится, например, единственный оператор *READ* .

3. Не пользуйтесь общей рабочей памятью для нескольких модулей. Нельзя знать заранее, в каком режиме работы операционной системы будет использоваться ваша программа; если таким режимом будет мультиобработка или режим разделения времени, то могут возникнуть конфликтные ситуации при использовании полностью независимыми модулями одних и тех же ячеек памяти для хранения промежуточных результатов (см. [I], стр. 29). Каждому модулю нужно отвести свою локализованную область памяти для хранения промежуточных данных, что означает, в частности, ограниченное использование, или даже полный отказ от применения областей *COMMON* в Фортране и описателя *EXTERNAL* в языке ПЛ/I.

4. Программируйте модули так, как вы программировали бы стандартные программы.

2.4. Структурное программирование

Мы рассмотрели основные этапы проектирования программы, теперь настала очередь непосредственного программирования модулей. К этому моменту ясно, ЧТО модуль должен делать, остается решить, КАК он это будет делать, иными словами, кодированию модулей предшествует выбор алгоритма.

Не начинайте программировать первый пришедший в голову алгоритм, а просмотрите, по крайней мере, несколько вариантов. Выберите лучший из них. Хороший алгоритм - необходимое (но не достаточное) условие получения хорошей программы. Если вы рассмотрите только один алгоритм для решения своей задачи, то вряд ли он окажется наилучшим. Всегда нужно иметь в виду, что час, потраченный на выбор алгоритма, стоит пяти часов программирования [2].

Существует немало литературных источников, в которых рассматриваются вычислительные алгоритмы: основные алгоритмы так называемого общего назначения хорошо представлены в трехтомнике Д.Кнута [7]; задачи линейной алгебры рассматриваются в справочнике Уилкинсона [8]; задачи нелинейного программирования хорошо освещены в книге Дж.Химмельблау [9] и т.д. Можно найти достаточное количество литературных источников, где представлены основные способы решения задач по любой тематике. Ознакомьтесь со всей доступной вам литературой по интересующему вопросу, возможно, что вы найдете уже готовый алгоритм, или даже готовую программу.

Как было отмечено в первой главе, одна из главных особенностей хорошей программы - удобочитаемость; были рассмотрены способы оформления листингов для повышения удобочитаемости программы. Вместе с тем и сам программный код должен быть удобочитаемым и легко анализироваться. Для достижения этой цели используется структурное программирование, или, как его еще называют, программирование без оператора *GOTO* . Истоки популярности и удобства структурированных программ следует искать в особенностях человеческого восприятия. Человек имеет склонность к последовательным процессам, будь то чтение детективной литературы или чужих программ. Чем больше разрывов в последовательности, тем труднее нам прослеживать логику повествования. После нескольких условных переходов мы можем потерять хронологию событий в детективной истории или запутаться в программе со сложным ветвлением, поэтому, чем меньше переходов, тем лучше.Например,попробуйте определить результат работы следую-

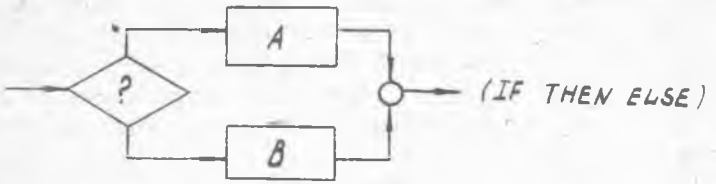
щей программы на ПЛ/I.

```
PRIMER: PROCEDURE OPTIONS(MAIN);
        GOTO L7;
L1: PUT EDIT('В')(A); GOTO L3;
L2: PUT EDIT('Р')(A); GOTO L4;
L3: PUT EDIT('Е')(A); GOTO L6;
L4: PUT EDIT('И')(A), GOTO L1;
L5: PUT EDIT('П')(A); GOTO L2;
L6: PUT EDIT('Т')(A); GOTO L8;
L7: PUT SKIP; GOTO L5;
L8:
END PRIMER;
```

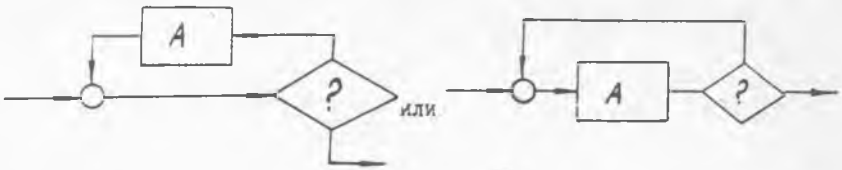
Структурное кодирование заключается в получении правильной программы из некоторых простых логических структур. Оно базируется на строгом доказательстве теоремы о структурировании [I], которая утверждает, что любую правильную программу с одним входом и одним выходом, без зацикливания и недостижимых команд можно написать с использованием только трех логических структур:

последовательности операторов

выбора одного из двух операторов:



повторения оператора или группы операторов, пока выполняется некоторое условие, или пока не достигнуто какое-то условие:



(DO WHILE) (DO UNTIL)

Применяя итерацию и вложение этих основных структур можно получить программу любого размера и сложности. Каждая структура прослеживается от начала до конца без каких-либо ветвлений.

Заметим, что операторами в этих структурах могут быть не только одиночные операторы, но и целые группы операторов инициируемые, например, при вызове подпрограммы, - важно лишь, чтобы они выполнялись последовательно.

Чтобы получить простую структуру для каждого фрагмента программы, следует избегать операторов *GOTO* . Каждый фрагмент программы должен состоять из последовательности операторов, операторов *IF THEN ELSE* (структура выбора), или циклов операторов. Некоторые языки программирования, например Фортран и Кобол, не имеют достаточного количества языковых конструкций для структурного программирования, поэтому при работе с ними трудно обойтись без операторов *GOTO* . В этом случае, а также для тех, кто не мыслит себе программирование без оператора *GOTO* , дадим две рекомендации по ограничению использования этого оператора:

1. Не прибегайте к *GOTO* там, где можно заменить его оператором *CALL* .

2. Применяйте *GOTO* только для переходов вперед. Переход назад подразумевает цикл (если нет, то это простая безалаберность), для его организации используйте соответствующую конструкцию цикла.

В языках ПЛ/1 и Алгол имеются необходимые средства для реализации рассмотренных логических структур. На Фортране труднее писать структурированные программы, так как в этом языке нет группирующих структур типа *IF THEN ELSE* .

Существует несколько способов реализации группы *IF* в Фортране. Покажем основные.

В случае одиночных операторов, которые нужно выполнить, лучше всего применить два последовательных оператора *IF* :

```
      IF  (условие) оператор 1
      IF  (.NOT. условие) оператор 2
```

Для выполнения нескольких операторов, определяемых оператором IF, можно сделать так:

```
      IF  (условие) GOTO 1ф
C     Выполнение, если условие ложно
      .
      .
      .
      GOTO 2ф
1ф ....
C     Выполнение, если условие истинно
      .
      .
      .
2ф    CONTINUE
```

Это, конечно, менее наглядно, чем при наличии оператора *IF THEN ELSE*, но делает те же самые действия. И здесь оператор *GOTO* используется только для перехода вперед. Для реализации остальных логических структур в Фортране имеются соответствующие языковые конструкции.

Руководствуясь принципами структурного программирования, можно существенно уменьшить сложность программ, сделать их понятными и легко анализируемыми. Структурированную программу можно читать сверху вниз как печатный текст.

Структурное программирование наиболее эффективно только в сочетании с нисходящим проектированием программ и модульным программированием. Однако хочется предостеречь от намеренного искажения программы только с той целью, чтобы исключить оператор *GOTO*, - любые крайности вредны. Следует понять, что структурное программирование не является самоцелью, его назначение - получение хорошей программы. Если вы будете следить за операторами *GOTO*, то в вашей программе они будут стоять именно там, где без них действительно не обойтись, и это не нарушит удобочитаемости программы, поскольку такие *GOTO* логически обоснованы.

В заключение отметим, что хотя и существуют основные принципы хорошего программирования, индивидуальный стиль программиста, образ его мышления и восприятия, творческий подход к делу всегда будут влиять на полученную программу.

3. ДОКУМЕНТИРОВАНИЕ ПРОГРАММ

"... Законы, если они не служат благу народа, суть лишь вздорные и ложные пророчества".

Ф.Бэкон. "О правосудии".

Для успешной разработки и внедрения математического обеспечения очень важна хорошая документация. К сожалению, зачастую составление документации к программам отстает от разработки самих программ, вследствие чего возникают достаточно серьезные трудности в выявлении причин отказа уже реализованной программы или при внесении в нее изменений. Кроме того, ряд методов ведения документации, не являющихся логическим продолжением методов разработки программ, осложняет составление необходимой документации.

Существуют государственные стандарты, регламентирующие форму итоговых документов по различным этапам создания программного обеспечения, а также ГОСТы на техническое задание по разработке программ, на рабочий проект программы, на описание и инструкцию по ее эксплуатации. Однако очевидно, что вся эта документация рождается не сама по себе, ей предшествует или должна предшествовать какая-то черновая, рабочая документация, которая никак не регламентирована, но оказывает огромное влияние на полноту и ясность перечисленных итоговых документов. Здесь представляется возможным обратиться к лучшему зарубежному опыту ведения подобной документации.

3.1. Методология ХИПО - универсальный способ документирования программ

В настоящее время документация к большей части программного обеспечения состоит из текста, схем и блок-схем. Блок-схемы полезны, но односторонни, так как отражают лишь один аспект программы – ее логику, т.е. КАК выполняются функции программы. Сами функции, т.е. ЧТО программа делает, при этом затушевываются. Сотрудники фирмы ИБМ считают, что если в документации особое внимание обраща-

ется на функциональный аспект, то сопровождение и эксплуатация системы будут более эффективными, поскольку при этом быстрее индентифицируется программный код, подлежащий модификации или исправлению. С этой целью фирмой ИБМ был разработан метод ХИПО - метод документирования функций.

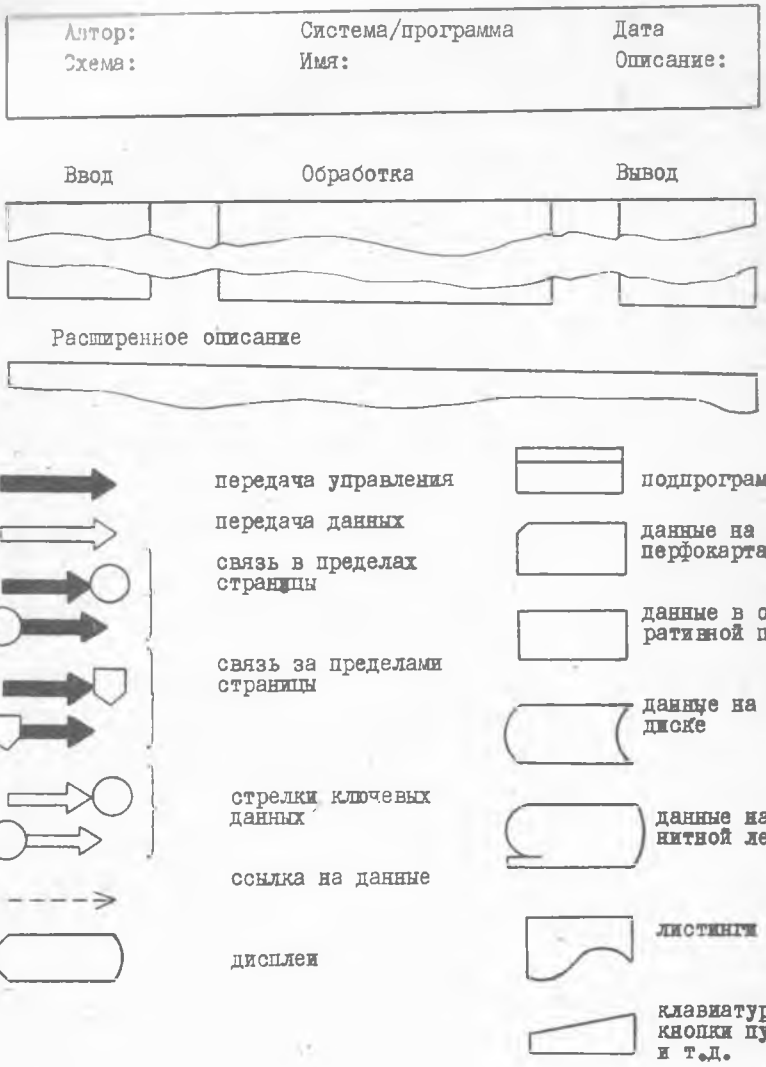
Метод ХИПО (аббревиатура английских слов Иерархия плюс Ввод-Обработка-Вывод) призван удовлетворить требования разработчиков, считающих, что документация должна иметь многоцелевое назначение. Например, администрации пользователя может понадобиться обзор системы; программисту-разработчику - документация, определяющая программные функции; программисту, занимающемуся сопровождением программы - документация, которая позволяет быстро найти программный код, подлежащий модификации или исправлению. Метод ХИПО способен удовлетворить все эти требования благодаря графическому представлению функций, выполняемых программой, приему постепенной детализации и указанию входных и выходных данных для каждого уровня.

Документация ХИПО состоит из набора схем, описывающих функцию программного обеспечения, начиная с общего и кончая самым детальным уровнем. Схема ХИПО представляет собой бланк, показанный на рис. I. В верхней части бланка записывают номер схемы, фамилию автора, название программы, к которой относится данная схема, имя программного модуля, изображенного на схеме, дату создания и функциональное назначение. Средняя часть бланка разделена на три колонки, в которых указывают: имена входных данных; функции обработки; имена выходных данных. Нижняя часть бланка отводится под расширенное описание структуры входных и выходных данных (спецификации), а также для пояснений вида обработки.

Разработка программы начинается с определения функций самого высокого уровня. Прежде всего, идентифицируется каждая основная функция, затем все они подразделяются на функции более низких уровней. Объединение функций низшего уровня соответствует одной функции высшего уровня.

Использование метода ХИПО с начала проектирования программы позволяет решить следующие основные задачи:

I. Разработать такую структуру программы и документации на нее, которая обеспечила бы понимание всех выполняемых функций. Схемы ХИПО должны иметь иерархическую структуру: каждая схема каждого уровня входит в схему вышележащего уровня.

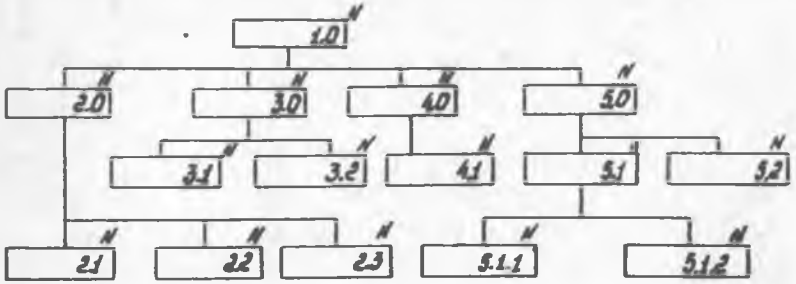

Автор:
Система/программа
Дата
Схема:
Имя:
Описание:
Ввод
Обработка
Вывод
Расширенное описание
передача управления
передача данных
связь в пределах страницы
связь за пределами страницы
стрелки ключевых данных
ссылка на данные
дисплеи
подпрограмма
данные на перфокартах
данные в оперативной памяти
данные на диске
данные на магнитной ленте
листинги
клавиатура, кнопки пульта и т.д.

Р и с. I

2. Определить функции, подлежащие выполнению, а не специфицировать программные операторы, которые используются для функций. Расширенное описание должно обеспечивать дополнительную информацию о функциях программы и призвано сократить объем ссылок на другую документацию.

3. Обеспечить визуально воспринимаемое описание входных и выходных данных для каждой функции каждого уровня схемы, разумеется, с разной степенью детализации. Основной целью каждой программы, если подходить с общих позиций, является преобразование входных данных в выходные, которые должны быть, во-первых, технически правильными, в во-вторых, удовлетворять требованиям пользователя. Метод ХИПО обеспечивает наглядное восприятие такого преобразования.

В типовую документацию ХИПО входят: наглядная таблица содержания документации, обзорные и подробные схемы (см.рис.I). Обзорные схемы по отношению к подробным более сокращенные, иногда в них даже отсутствует расширенное описание. Наглядная таблица содержания документации представляется в виде иерархической структуры (рис.2).



Р и с. 2

В каждом прямоугольнике указывается название соответствующей схемы и ее номер, вверху прямоугольника - номер страницы, на которой расположена схема. Наглядная таблица содержания документации отражает структуру документации и одновременно связь между функциями в разрезе иерархической организации. Имея наглядную таблицу содержания документации, можно легко найти нужный уровень информации или конкретную схему.

Схемы высокого уровня, или обзорные схемы, описывают основные функции и содержат ссылки на подробные схемы, обеспечивающие требуемую детализацию функций.

В подробных схемах ХИПО содержатся основные элементы документации, описываются конкретные функции, приводятся конкретные входные и выходные данные, даются ссылки на другие подробные схемы. В разделе "Расширенное описание" приводится дополнительная информация об этапах обработки и дается ссылка на программный код, соответствующий конкретному этапу. Количество уровней подробных схем определяется количеством функциональных блоков, сложностью материала и количеством информации, подлежащей документированию.

Существует три типа документации ХИПО: документация начальной разработки программы, детальной разработки и документация для сопровождения, в каждом из видов представлены схемы трех типов, но документация каждого типа отличается целями, характеристиками и контингентом пользователей (рис.3).

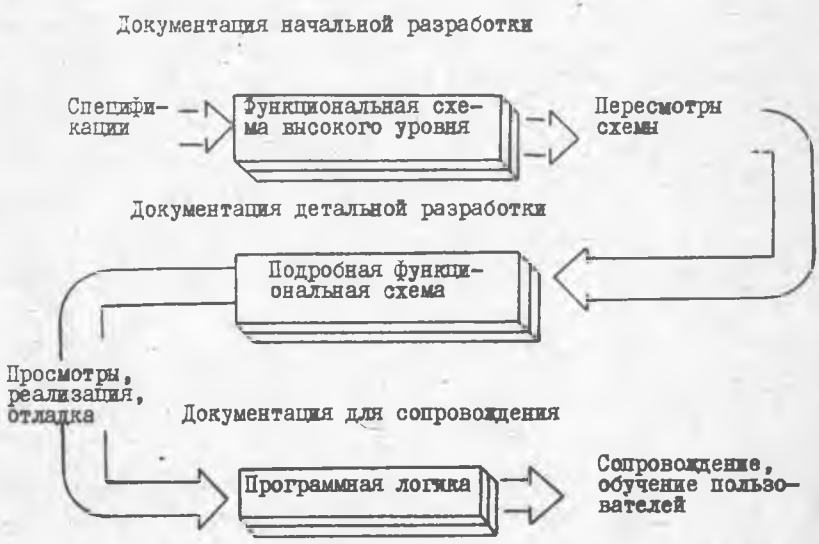


Рис. 3

Документация начальной разработки составляется группой, разрабатывающей общую концепцию программы в самом начале работы над проектом, содержит описание общего

функционального построения программы и используется как вспомогательное средство для проектирования. Основу документации составляет техническое задание на программу или постановка задачи (см. пп. 2.1). Разработчики пользуются схемами ХИПО для того, чтобы обменяться предложениями и проверить их, затем этой же документацией пользуется заказчик при просмотре программы, а также для утверждения основных спецификаций и постановки задачи.

Документация детальной разработки составляется группой непосредственных разработчиков программы. Используя в качестве основы документацию начальной разработки, программисты вырабатывают подробные спецификации и прибавляют к подробным схемам ХИПО дополнительные уровни. Сформированным документом пользуются при внедрении программы и для сравнения с документацией начальной разработки с целью проверки соблюдения всех требований. В процессе кодирования программы дополняют расширенное описание: детализируют строение входных и выходных данных, указывают метки программы и приводят информацию, необходимую для внедрения программы. По окончании этой работы программисты имеют в своем распоряжении полный, точный и удобный для восприятия документ, являющийся основой для работы людей, отвечающих за сопровождение программы, и для составления инструкций пользователям. Зачастую документация детальной разработки представляет собой окончательный документ и используется при сопровождении программы.

Составленная отдельно документация для сопровождения представляет собой документацию детальной разработки, из которой изъят ряд схем низших уровней, содержащих детальную информацию для кодирования, в результате чего значительно сокращается потребность в разработке дополнительных схем ХИПО для сопровождения.

При разработке программ важно как можно раньше начать применять метод ХИПО с тем, чтобы разработчики могли документировать свои соображения и действия в ПРОЦЕССЕ создания программы, а также и для того, чтобы функции, которые должна выполнять программа, не затерялись по ходу работы. Зачастую соображения разработчиков не приводятся к формализованному виду и никак не документируются - это ведет к непродуктивному использованию времени, реализации недостаточно понятых функций - в итоге получается программа, которая не может удовлетворить все запросы пользователей, сформулированные в техническом задании.

Составление схем ХИПО - это не дополнительная нагрузка, а естественный побочный продукт процесса создания программного обеспечения. В любом случае к определенному моменту разработки программы должны быть определены потребности и функции, а также соответствующие вводы и выводы. Эта информация тоже зачастую не документируется, ее обмениваются устно, или, в лучшем случае, записывают в примечаниях либо в черновиках, которые не включают в итоговые документы. Метод ХИПО обеспечивает запись в наглядной форме и доведение такой информации до всех заинтересованных лиц.

Теперь на конкретном примере проектирования прикладной программы рассмотрим все этапы использования метода ХИПО.

3.I.I. Документирование на стадии проектирования. Пусть требуется создать небольшую прикладную программу расчета напряжений в стержнях произвольной фермы и определения силового веса [I0] материала ее конструкции. Первым этапом явится полная и подробная разработка пользовательских спецификаций, т.е. получение наиболее полного ответа на вопрос: "Что мы хотим получить от создаваемой программы?". Прежде всего нужно определить назначение программы и область ее применения.

В нашем случае программа создается для использования ее в лабораторной работе по отысканию оптимальной фермы. Пользователь должен вводить с терминала исходные данные, а программа - рассчитывать напряжения в стержнях и вычислять силовой вес материала конструкции как меру оптимальности фермы. После анализа результатов расчета пользователь может изменить либо взаимное расположение стержней фермы - т о п о л о г и ч е с к у ю с х е м у, либо координаты одного или нескольких узлов - г е о м е т р и ю, либо распределение материала по стержням - ж е с т к о с т и. Предполагается одновременная работа не менее двух пользователей, лабораторная работа проводится в форме соревнования: кто за определенное время получит конструкцию с меньшим значением силового веса?Теперь нужно выявить наличие информации, необходимой для выполнения этой функции, т.е. следует ответить на вопрос: "Возможно ли рассчитать произвольную ферму, и если да, то каким методом и при каких исходных данных?".

Из строительной механики и теории упругости мы знаем, что сейчас широко применяется универсальный метод расчета произвольных конструкций - метод конечных элементов [II,I2]. Для его использования применительно к фермовым конструкциям необходимо:

пронумеровать отдельно узлы и стержни фермы;

определить топологию конструкции, т.е. с какими узлами связан каждый стержень. Узел идентифицируется своим номером;

определить геометрию конструкции, т.е. координаты всех узлов фермы;

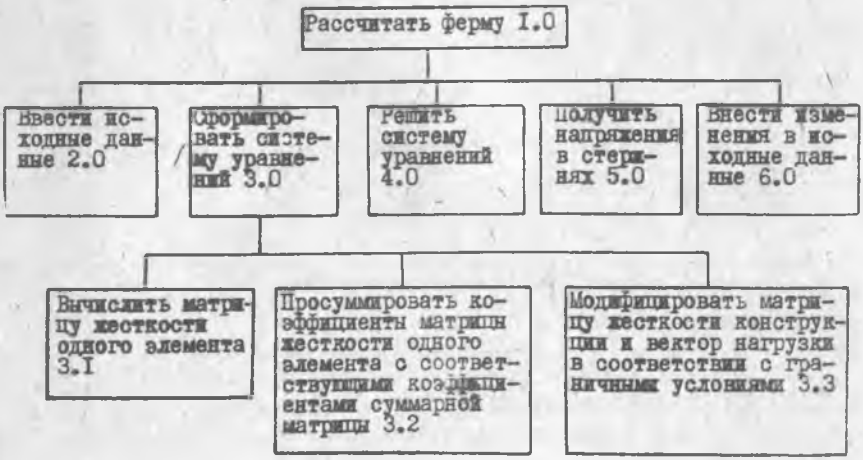
задать площадь поперечных сечений и характеристики материала стержней;

задать нагрузки, действующие в узлах фермы;

задать граничные условия (закрепления) для узлов фермы.

Таким образом, содержательная часть входной информации нам известна, известен также метод, с помощью которого на ее основе можно получить выходную информацию - напряжения в стержнях и величину силового веса.*

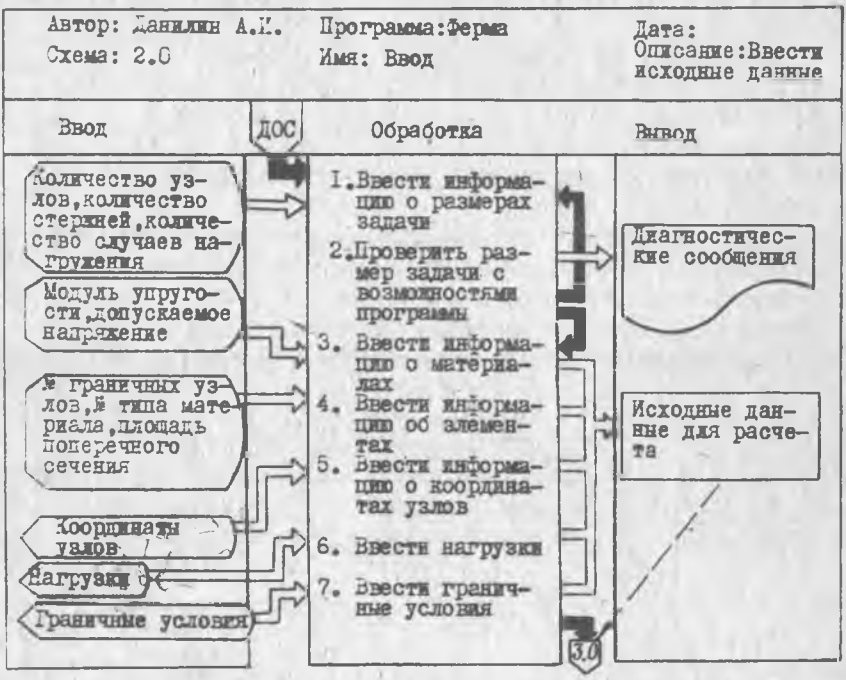
Теперь необходимо полностью определить функции, подлежащие выполнению программой, и их взаимосвязь. Определение функций лучше всего сделать в форме наглядной таблицы (рис.2) (заметьте, что одновременно с разработкой программы мы создаем и документацию на нее!). Последовательность решения задачи показана на рис. 4. При составле-



Р и с. 4

* Силовой вес для ферм вычисляется как $\sum_{i=1}^{m} |\sigma_i| F_i l_i$, где m - количество стержней, $|\sigma_i|$ - абсолютная величина напряжения, F_i - площадь поперечного сечения, а l_i - длина стержня.

нии наглядной таблицы нужно руководствоваться одним общим правилом: расчленение на низшие уровни нужно проводить до тех пор, пока и разработчикам, и заказчику не будут полностью понятны описываемые функции. Параллельно с составлением наглядной таблицы для каждой из функций, указанных в четырехугольниках верхнего уровня на рис.4, разрабатываются обзорные схемы ХИПО, показанные на рис. 5, 6.



Расширенное описание

Все исходные данные хранятся в оперативной памяти и используются другими блоками программы. Форма хранения исходных данных, а также входной формат должны быть определены при составлении подробных схем.

Рис. 5

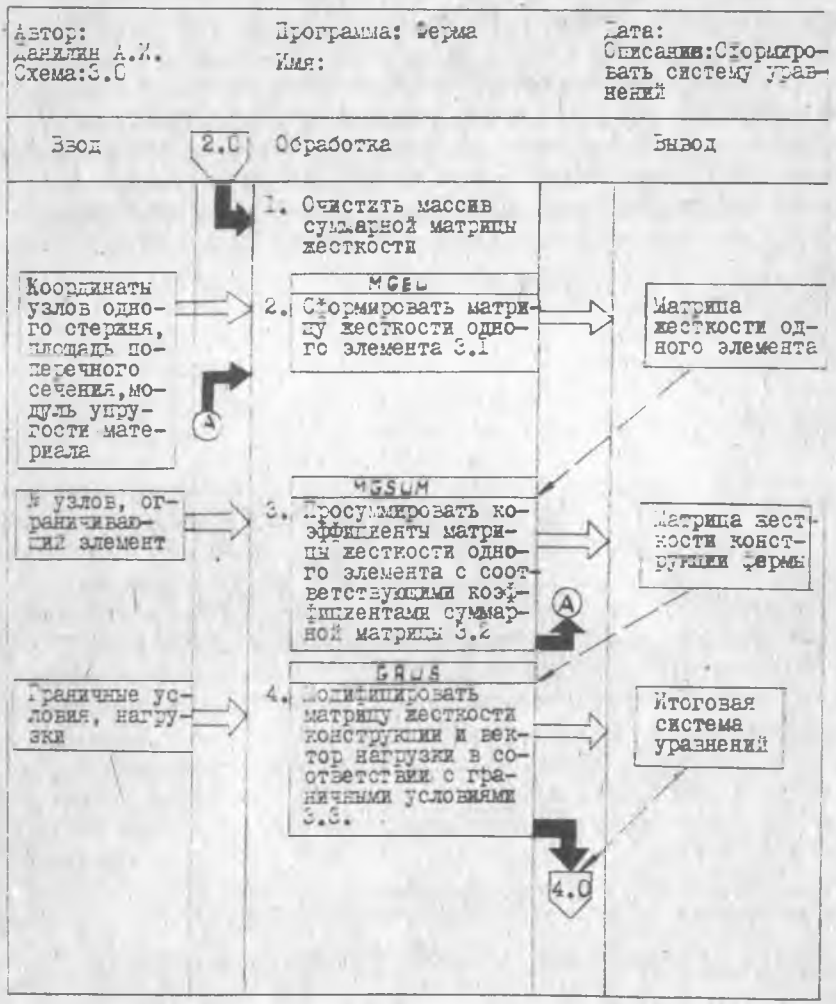


Рис. 6

Обзорная схема должна давать общее представление о конкретной функции. В ней указываются все основные вводы-выводы, а сама она является руководством для схем более низких уровней. Хотя на этом эта-

не необязательно идентифицировать конкретные устройства ввода-вывода, но, когда они известны, их можно указать с помощью соответствующих обозначений.

Далее программистам-разработчикам предстоит составить подробные схемы ХИПО (рис. 7). Основное внимание при этом уделяется вводу-выводу, а обработка опять-таки описывается только в функциональном аспекте. Формат входных и выходных данных подробно дается в расширенном описании; если схема ХИПО не умещается на одном листе, то она может быть продолжена далее на нужное количество листов.

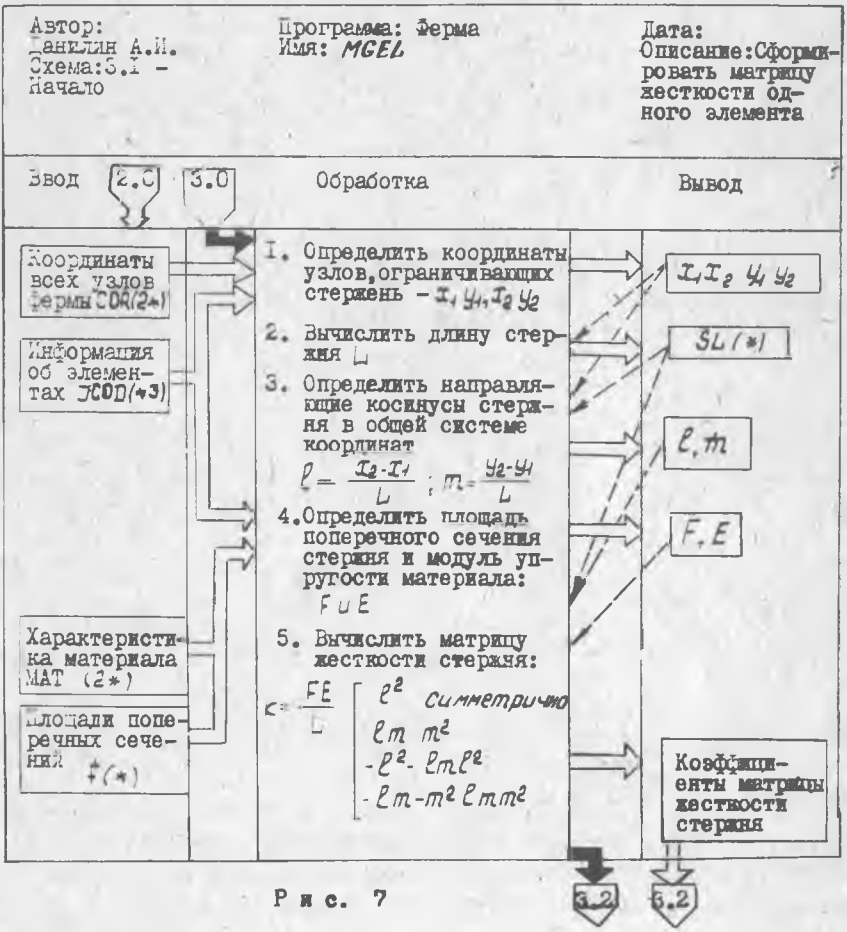


Рис. 7

Расширенное описание

COR(2,*)- координаты всех узлов фермы. COR(1,i)- координата по оси X ; COR(2,i)- по оси Y ; i - номер узла. Значения координат даются в [мм] .

JCOD(*,3) - информация об элементах. JCOD(i,1)- номер первого узла стержня; JCOD(i,2)- номер второго узла; JCOD(i,3)- номер типа материала; i - номер элемента.

MAT(2,*) - характеристики материала. Тип данных - REAL

MAT (1,j) - модуль упругости [МПа] ; MAT(2,j) - допускаемое напряжение [МПа] ; j - номер типа материала.

Для поиска характеристик материала стержня i следует выполнить действие: MAT (..., JCOD(i,3)).

Для поиска координаты X первого узла i -го стержня нужно выполнить: COR(1, JCOD(i,1)); координаты Y : COR(2JCOD(i,1)) . Для поиска координат второго узла используется JCOD(i,2).

F(*)- площади поперечных сечений стержней [мм^2]. Значения площади i стержня равно F(i).

Длины стержней копятся в массиве SL(i), i - номер стержня.

Неописанные типы данных соответствуют типам данных по умолчанию.

Коэффициенты матрицы жесткости стержня хранятся в виде скалярных переменных:

PL соответствует (FE/L)e^2; PM- (FE/L)$\cdot m^2$;

PLM-(FE/L)em . В дальнейшем они рассылаются на соответствующие места в суммарной матрице жесткости со своими знаками.

Р и с. 7 (продолжение)

На первых этапах работы по методу ХИПО трудно:

изложить идею сжато;

решить, какие функции можно расчленить на подфункции;

определить оптимальную разбивку на уровни иерархии. Часто пытаются на одной схеме записать как можно больше, вместо того, чтобы составить схемы более высоких уровней;

давать формулировки в разрезе функций, а не с точки зрения программной логики.
Преодолеть все эти трудности возможно опытным путем: нужно ошибаться, стирать, переделывать, вновь ошибаться и так до тех пор, пока исчезнут затруднения.

Главное правило, которым всегда нужно руководствоваться при составлении схем, следующее: схемы ХИПО должны показывать ЧТО программа делает, а не КАК она работает. В результате применения метода ХИПО становится возможным просмотреть сделанное значительно раньше, чем при составлении блок-схем и текстовых комментариев, поскольку разработку программы и составление документации выполняют одновременно и постепенно по уровням, начиная с самого общего уровня и далее детализируя. Просмотры программы на этапе начальной разработки желательно проводить совместно с заказчиком (пользователем), причем для этого совсем не обязательны окончательные варианты схем ХИПО, важно только, чтобы схемы были понятными.

Участники просмотров прежде всего обращают внимание на: ф у н к ц и о н а л ь н у ю т о ч н о с т ь: верно ли определена функция? Нужно ли что-нибудь добавить или опустить? Верно ли определен необходимый ввод и вывод?

Т е х н и ч е с к у ю т о ч н о с т ь: ясно ли показано, какие конкретные выходные данные формируются на данном этапе? Конкретно ли описание функции? Учтены ли все запросы пользователя? Участники просмотров работают с документацией самостоятельно, не прибегая к помощи ее составителей. Если требуются какие-либо объяснения, то это означает, что документация нуждается в доработке.

В нашем случае в результате проведения просмотров наглядной таблицы и обзорных схем выявились два упущения, сделанные при постановке задачи: неясно, на сколько случаев нагружения должна быть оптимизирована ферма - на один или несколько? Может ли пользователь за терминалом сразу изменять несколько типов исходных данных, например: одновременно топологию и жесткости? Естественно, эти вопросы были согласованы с заказчиком, который после изучения документации начальной разработки программы уже точно знает, что же он получит в итоге.

3.1.2. Документирование для кодирования и отладки. На этапе детальной разработки и внедрения программы предполагается:

расширить документацию до такой степени детализации, чтобы на ее основе можно было написать программный код;

поставить в соответствие функциям модули программы;

разработать программный код;

разработать и выполнить контрольные примеры;

составить инструкции для операторов.

Вводом на данном этапе служит документация проектирования программы. Расширяются имеющиеся и составляются более подробные схемы с такой степенью детализации, чтобы на их основе можно было писать программный код, однако к излишней детализации, так что одной функции соответствовал бы, к примеру, один оператор – стремиться не следует. Если проектирование программы проведено достаточно тщательно и подробно, то дорабатывать документацию придется незначительно, обычно такая доработка состоит в следующем:

присваиваются конкретные имена модулям, массивам, полям, таблицам, переключателям и т.д.;

конкретизируются устройства ввода-вывода. Уточняется формат промежуточных данных с целью максимального соответствия виду обработки;

в расширенное описание включается: дополнительная информация об обработке; дополнительные детали для идентификации данных; программные метки для кода; ссылки на схемы ХИПО более низких уровней и на документацию, составленную не по методу ХИПО.

При проведении этапа кодирования программы важно постоянно иметь в виду, что:

имена, присвоенные программам, файлам, элементам данных, меткам и т.д., должны устойчиво сохраняться на всех схемах как документации проектирования программы (если они там определены), так и документации для кодирования (здесь они должны быть обязательно). Особое внимание следует обратить на стыковку всех вводов и выводов;

все подробные схемы должны являться поднаборами схем более высокого уровня, причем это относится как к функциональному содержанию, так и к основным вводам и выводам;

подробные схемы документации для кодирования следует сравнивать с подробными схемами документации проектирования программы с тем, чтобы проверить, ничего ли не пропущено или незаконно изменено, ничего ли не добавлено;

разработанный код каждого модуля следует сравнивать с соответствующей подробной схемой с тем, чтобы убедиться в реальности исполнения функций

В целом документация для кодирования призвана предстать ясной, точной и подробной информацией о программе для ее дальнейшего сопровождения, к ней обязательно прилагаются листинг программы, описание контрольных примеров и листинги результатов их выполнения, кроме того, допускается приложение любой другой документации, выполненной не по методу ХИПО.

Теперь обратимся к примеру создания программы расчета ферм. Внимательное рассмотрение подробной схемы (см.рис.7) показывает, что описанные функции можно непосредственно кодировать, например, на Фортране, без какой-либо дополнительной детализации. Если текст программы на Фортране снабжен комментариями о названиях функций (см.рис.7), то схема вообще не нуждается в доработке, поскольку происходит полная идентификация функций и программного кода, с помощью которого они реализуются. Если же такие комментарии не будут вставлены, то на схеме для каждой из функций указывают соответствующие номера операторов Фортрана или их метки.

3.I.3. Документирование для сопровождения программы. Документацию для сопровождения программы получают из документации для кодирования механическим изъятием ряда схем самого низкого уровня. Необязательное правило здесь гласит: если схема ХИПО соответствует 5-20 операторам исходного текста программы, то ее можно изъять и пользоваться схемой более высокого уровня.

Определяя пригодность документации для кодирования при сопровождении программы документацию просматривают с привлечением программистов, которые будут заниматься сопровождением, при этом особое внимание обращают на то:

полна ли и ясна документация? Можно ли проследить все логические коды, понятны ли они?

Понятны ли из схем ХИПО цели функций, отраженных в коде?

Читабелен ли исходный текст программы?

Насколько подробны схемы самого низкого уровня? Если принимается решение об изъятии схемы, то проверяют схему предыдущего уровня с тем, чтобы быть уверенным, что в схеме и в расширенном описании представлена вся необходимая информация, в частности, есть ссылки на программный код.

Заключение

Мы рассмотрели все основные этапы создания программы, выявили отличительные особенности хороших программ и наметили пути достижения этих свойств. Теперь настала пора практического использования высказанных рекомендаций.

Если вы пишите трудночитаемые, неструктурированные малоэффективные программы и при этом неудовлетворены своей работой, то попробуйте применить на практике материал данного пособия. Мы уверены, что ваша продукция существенно улучшится и вы останетесь довольны результатами своего труда.

Если же вы обладаете блестящими способностями и для вас нет тайн в программировании, надеемся, что все изложенные соображения помогут вам совершенствовать свое мастерство.

Литература

1. Йодан Э. Структурное проектирование и конструирование программ. - М.: Мир, 1979. - 415с.
2. Ван Тассел Д. Стиль, разработка, эффективность, отладка и испытание программ. - М.: Мир, 1981. - 319с.
3. Брукс Ф.П. Как проектируются и создаются программные комплексы. Мифический человеко-месяц. Очерки по системному программированию. - М.: Наука, 1979. - 250с.
4. *D.E. Knuth. An Empirical Study of FORTRAN Programs, Software - Practice and Experience, v.1, N2, April - June, 1971, p. 105 - 133.*
5. *Morrison J.E. User Performance in Virtual Storage Systems. IBM System Journal, No. 3 (1973).*
6. *Wulf W.A. A Case Against the GOTO. Proceeding of the 25th ACM National Conference, v.2, 1972, p. 791 - 797.*
7. Кнут Д. Искусство программирования для ЭВМ. - М.: Мир, т.1, 1976. -735с., т.2, 1977. -724с., т.3, 1978, -844 с.
8. Уилкинсон, Райнш. Справочник алгоритмов на языке Алгол. Линейная алгебра. - М.: Машиностроение, 1976. - 390с.

9. Химмельблау Дж. Прикладное нелинейное программирование. - М.: Мир, 1975. - 534с.
10. Комаров А.А. Основы проектирования силовых конструкций. - Куйбышев: Куйбышевское книжное изд-во, 1965. -88с.
11. Зенкевич О. Метод конечных элементов в технике. -М.:Мир, 1975. -541с.
12. Хазанов Х.С., Савельев Л.М. Метод конечных элементов в приложении к задачам строительной механики и теории упругости: Конспект лекций, п.4.1. - Куйбышев: Куйбышевский авиационный ин-т, 1975. - 128с.
13. *HIPO - A Design Aid and Documentation Technique. Installation Managment Sezies GC 20-1851, Machanicsburg: Inteznational Business Machines, 1974.*

О Г Л А В Л Е Н И Е

Св. план I983, поз.25

Александр Иванович Д а н и л и н

ПРИКЛАДНОЕ МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ САПР.
МЕТОДЫ ПРОЕКТИРОВАНИЯ И ДОКУМЕНТИРОВАНИЯ ПРОГРАММ

Учебное пособие

Редактор Е.Д. А н т о н о в а
Техн.редактор Н.М. К а л е н ю к
Корректор Н.С. К у п р и я н о в а

Подписано в печать 26.05.83 г. ЕО 00183.
Формат 60х84 I/I6. Бумага оберточная белая.
Оперативная печать. Усл.п.л. 3,2. Уч.-изд.л. 3,0.
Тираж 500 экз. Заказ 3468 Цена I0 к.

Куйбышевский ордена Трудового Красного Знамени авиационный институт им.академика С.П.Королева, г. Куйбышев, ул. Молодогвардейская, I5I.

Типография им. В.П.Мяги, г. Куйбышев,ул.Венцека,60.