М. Бен-Ари Языки программирования. Практический сравнительный анализ. Предисловие

Значение языков программирования

Сказать, что хороший программист может написать хорошее программное обеспечение на любом языке, — это все равно, что сказать, что хороший пилот может управлять любым самолетом: верно, но не по существу. При разработке пассажирского самолета основными критериями являются безопасность, экономическая целесообразность и удобства; для военного самолета главное это летные качества и возможность выполнения боевой задачи; а при создании сверхлегкого самолета необходимо обеспечить низкую стоимость и простоту управления.

Роль языка в программировании принижается по сравнению с программной методологией и инструментальными средствами; и не только преуменьшается, но и полностью отвергается, когда утверждают, что хорошо разработанная система может быть одинаково хорошо реализована на любом языке. Но языки программирования — это не просто инструментальное средство;
это тот «материал», из которого создается программное обеспечение, то, что мы видим на наших экранах большую часть дня. Я верю, что язык программирования — один из наиболее, а не наименее важных факторов, которые влияют на окончательное качество программной системы. К сожалению, слишком у многих программистов нет достаточных языковых навыков. Они страстно любят свой «родной» язык программирования и не способны ни проанализировать и сравнить конструкции языка, ни оценить преимущества и недостатки современных языков и языковых понятий. Слишком часто можно услышать утверждения, демонстрирующие концептуальную путаницу: «Язык L1мощнее (или эффективнее) языка L2».

С этим недостатком знания связаны две серьезные проблемы разработки программного обеспечения. Первая — крайний консерватизм в выборе языков программирования. Несмотря на бурное развитие компьютерной техники и сложности современных программных систем, большинство программ все еще пишутся на языках, которые были разработаны около 1970 г., если
не раньше. Многие исследования в области языков программирования никогда не подвергались проверке практикой, и разработчики программ вынуждены с помощью различных инструментальных средств и методологий компенсировать устаревшую языковую технологию. Это примерно то же, что , отказ авиакомпании испытать реактивный лайнер на том основании, что старые винтомоторные самолеты и так могут прекрасно доставить вас куда нужно.

Вторая проблема состоит в том, что языковые конструкции используются без должного отбора, практически без учета надежности и эффективности. Это ведет к созданию ненадежного программного обеспечения, которое невозможно поддерживать, а также к неэффективности, которая устраняется скорее путем кодирования отдельных фрагментов программ на языке ассемблера, чем совершенствованием алгоритмов и парадигм программирования.

Языки программирования существуют *только* для преодоления разрыва в уровне абстракции между аппаратными средствами и реальным миром. Есть неизбежное противоречие между высшими уровнями абстракции, которые легче понять и безопаснее использовать, и низшими уровнями, более гибкими и зачастую допускающими более эффективную реализацию. Чтобы разработать или выбрать язык программирования, следует избрать соответствующий уровень абстракции, и нет ничего удивительного в том, что разные программисты предпочитают различные уровни и что какой-либо язык может подходить для одного проекта и не подходить для другого. Программисту следует хорошо понимать степень надежности и эффективности каждой конструкции языка.

Цель книги

Цель этой книги — научить читателя разбираться в языках программирования, анализируя и сопоставляя языковые конструкции, и помочь ему уяснить:

- Какие альтернативы доступны разработчику языка?
- Как реализуются языковые конструкции?
- Как их следует использовать?

Мы, не колеблясь, заявляем: накопленный опыт показывает, что одни конструкции предпочтительнее других, а некоторых следует избегать или, по крайней мере, использовать их с осторожностью.

Конечно, эту книгу не следует рассматривать как справочник по какому-либо конкретному языку программирования. Задача автора заключается в том, чтобы научить анализировать языки, не погружаясь в мелкие языковые частности. Книга также не является руководством по выбору языка для какого-либо конкретного проекта. Цель состоит в обеспечении учащегося концептуальными инструментальными средствами, необходимыми для принятия такого решения.

Выбор материала

Автору книги по языкам программирования неизбежно приходится обижать, по крайней мере 3975 из 4000, если не больше, изобретателей различных языков! Я сознательно решил (даже если это обидит 3994 человека) сосредоточить внимание на *очень* небольшом наборе языков, поскольку уверен, что на их примере смогу объяснить большинство языковых понятий. Другие языки обсуждаются только при демонстрации таких понятий, которые отсутствуют в языках, выбранных для основного рассмотрения.

Значительная часть книги посвящена «заурядным» процедурным (императивным, imperative) языкам; из этого класса выбраны два. Языки с низким уровнем абстракции представляет C, который обошел Fortran, прежде доминирующий в этой категории. Для представления более высокого уровня абстракции мы выбрали язык Ada с гораздо более четкими определениями, чем в широко известном языке Pascal.

Этот выбор оправдывает также то, что оба языка имеют расширения (C++ и Ada 95), которые можно использовать для изучения языковой поддержки объектно-ориентированного метода программирования, доминирующего в настоящее время.

К сожалению, (как я полагаю) большинство программ сегодня все еще пишутся на процедурных языках, но за последние годы качество реализаций непроцедурных (неимперативных) языков улучшилось настолько, что они могут использоваться для разработки «реального» программного обеспечения. В последних главах представлены функциональные (ML) и логические (Prolog) языки программирования с целью убедить учащихся, что процедурные языки не являются концептуальной необходимостью для программирования.

Теория синтаксиса языков программирования и семантики выходит за рамки этой книги. Эти важные предметы лучше оставить для более продвинутых курсов.

Чтобы избежать путаницы при сравнении примеров на разных языках, каждый пример сопровождается обозначением типа C++ . В разделах, где обсуждаются конструкции определенного языка, обозначения не даются.

О чем эта книга

Часть 1 является описательной. Она содержит определения и обзор языков и сред программирования. Во второй части подробно объясняются основные конструкции языков программирования: типы, операторы и подпрограммы. В части 3 рассматриваются более сложные понятия программирования, такие, как действительные числа, статический полиморфизм, обработка ошибок и параллелизм. В части 4 обсуждается программирование больших систем с акцентом на языковой поддержке объектно-ориентированного программирования. Заключительная часть 5 посвящена основным концепциям функционального и логического программирования.

Рекомендации по обучению

Необходимое условие для изучения этой книги — по крайней мере один год программирования на каком-либо языке типа Pascal или C. В любом случае, студент должен уметь читать C-программы. Также будет полезно знакомство со структурой и набором команд какого-либо компьютера.

На основе изложенного материала можно составить несколько курсов лекций. Части 1 и 2 вместе с разделами части 4 по модулям и объектно-ориентированному программированию могут послужить основой односеместрового курса лекций для второкурсников. Для продвинутых студентов можно ускорить изложение первой половины, с тем чтобы сосредоточиться на более трудном материале в частях 3 и 4. Углубленный курс, несомненно, должен включить часть 5, дополненную в большом объеме материалом по некоторому непроцедурному языку, выбранному преподавателем. Разделы, отмеченные звездочкой, ориентированы на продвинутых студентов.

Для большинства языков можно бесплатно получить компиляторы, как описано в приложении A. Студенты также должны быть обучены тому, как просмотреть команды ассемблера, генерируемые компиляторами.

Упражнения: поскольку эта книга о языках программирования, а не по программированию, то в упражнениях не делается акцент на проектировании программ. Вместо этого мы просим студентов покопаться в описаниях, сравнить языки и проанализировать, как компилятор реализует различные конструкции. Преподаватель может изменить упражнения и добавить другие согласно своему вкусу и доступности инструментальных средств.

Книга будет также полезна программистам, которые хотят углубить свои знания об инструменте, которым они ежедневно пользуются, — о языках программирования.

Примечание автора

Лично я предпочитаю более высокие уровни абстракции низким. Это — убеждение, а не предубеждение. Нам — разработчикам программного обеспечения — принадлежат печальные рекорды в вопросах разработки надежных программных систем, и я полагаю, что решение *отчасти* лежит в переходе к языкам программирования более высоких уровней абстракции. Обобщая высказывание Дейкстры, можно утверждать: если у вас есть программа в 100 000 строк, в которой вы запутались, то следует переписать ее в виде программы в 10 000 строк на языке программирования более высокого уровня.

Первый опыт я получил в начале 1970-х годов как член большой группы программистов, работающих над системой финансовых транзакций. Мы установили новую интерактивную систему, хотя знали, что она содержала ошибку, которую мы не могли найти. Спустя несколько недель ошибка была, наконец, обнаружена: оказалось, что изъяны в используемом языке программирования привели к тому, что тривиальная опечатка превратилась в несоот-

ветствие типов. Пару лет спустя, когда я впервые увидел Pascal, меня «зацепило». Мое убеждение в важности проблемы усиливалось всякий раз, когда я помогал ученому, потратившему впустую недели на отыскание ошибки в программе, причем в такой, которую, будь она на языке Pascal, нельзя было бы даже успешно *скомпилировать*. Конечно, несоответствие типов — не единственный источник ошибок программирования, но оно настолько часто встречается и так опасно, хотя и легко обнаруживается, что я считаю жесткий контроль соответствия типов столь же необходимым, как и ремень безопасности в автомобиле: использование его причиняет неудобство, но оно весьма незначительно по сравнению с возможным ущербом, а ведь даже самые лучшие водители могут попасть в аварию.

Я не хочу быть вовлеченным в языковые «войны», утверждая, что один язык лучше другого *для какой-либо определенной* машины или прикладной программы. Я попытался проанализировать конструкции языка по возможности объективно в надежде внести вклад в повышение уровня научных дискуссий относительно языков программирования.

Благодарности

Я хотел бы поблагодарить Кевлина А.П. Хеннея (Kevlin A.P Henney) и Дэвида В. Баррона (David W. Barron) за ценные замечания по всей рукописи, так же как Гарри Майрсона (Harry Mairson), Тамара Бенея (Tamar Benaya) и Бруриа Хабермена (Bruria Haberman), которые прочитали отдельные части. Я обязан Амирему Ехудаи (Amiram Yehudai), моему гуру в объектно-ориентированном программировании: он руководил мной во время многочисленных обсуждений и тщательно проверял соответствующие главы. Эдмон Шенберг (Edmond Schonberg), Роберт Девар (Robert Dewar) вместе со своей группой в NYU быстро отвечали на мои вопросы по GNAT, позволив мне обучиться и написать о языке Ada 95 еще до того, как стал доступен полный компилятор. Ян Джойнер (lan Joyner) любезно предоставил свой неопубликованный анализ языка C++, который был чрезвычайно полезен. Подобно моим предыдущим книгам, эта, вероятно, не была бы написана без LATEX Лесли Лампорта (Leslie Lamport)!

Мне посчастливилось работать с высоко профессиональной, квалифицированной издательской группой Джона Уайли (John Wiley), и я хотел бы поблагодарить всех ее членов и особенно моего редактора Гейнора Редвеса-Мат-тона (Gaynor Redvers-Mutton).

М. Бен-Ари

Реховот, Израиль

1 Введение в языки программирования

Глава 1
Что такое языки программирования

1.1. Некорректный вопрос

Первый вопрос, который обычно задает человек, впервые сталкивающийся с новым языком программирования:

Что этот язык может «делать»?

Неявно мы сравниваем новый язык с другими. Ответ очень прост: все языки могут «делать» одно и то же — производить вычисления! В разделе 1.8 объяснена правомерность такого ответа. Однако, если все они могут выполнять одно и то же — вычисления — то, несомненно, причины существования сотен языков программирования должны быть в чем-то другом.

Позвольте начать с нескольких определений:

Программа — это последовательность символов, определяющая вычисление.

Язык программирования — это набор правил, определяющих, какие последовательности символов составляют программу и какое вычисление описывает программа.

Вас может удивить, что в определении не упоминается слово «компьютер»! Программы и языки могут быть определены как сугубо формальные математические объекты. Однако люди больше интересуются программами, чем другими математическими объектами типа групп, именно потому, что программу — последовательность символов — можно использовать для управления работой компьютера. Хотя мы настоятельно рекомендуем изучение теории программирования, здесь ограничимся, в основном, изучением того, *как программы выполняются* на компьютере.

Эти определения следует понимать в самом широком смысле. Например, сложные текстовые процессоры обычно имеют средство, которое позволяет запоминать последовательность нажатий клавиш и сохранять их как *макрос*, чтобы всю последовательность можно было выполнять с помощью единственного нажатия клавиши. Несомненно, это — программа, поскольку последовательность нажатий клавиш определяет

вычисление, и в сопроводительной документации обязательно будет определен макроязык: как инициализировать, завершать и называть макроопределение.

Чтобы ответить на вопрос, вынесенный в название главы, вернемся к первым цифровым компьютерам, очень похожим на простые калькуляторы, какими сегодня пользуются для расчетов в магазине. Они работали по «жесткой» программе, которую нельзя изменить.

Наиболее значительным из первых шагов в усовершенствовании компьютеров была идея (автором которой считается Джон фон Нейман) о том, что описание вычисления (программу) можно *хранить* в памяти компьютера так же, как данные. *Компьютер с запоминаемой программой,* таким образом, становится универсальной вычислительной машиной, а программу можно изменять, только заменяя коммутационную доску, вводя перфокарты, вставляя дискету или подключаясь к телефонной линии.

Поскольку компьютеры — двоичные машины, распознающие только нули и единицы, то хранить программы в компьютере технически просто, но практически неудобно: каждая команда должна быть записана в виде двоичных цифр *(битов),* которые можно представить механически или электрически. Одним из первых программных средств был *символический ассемблер.* Ассемблер берет программу, написанную на *языке ассемблера* (каждая команда представлена в нем в символьном виде), и транслирует символы в двоичное представление, пригодное для выполнения на компьютере. Например, команду

```
load    R3,54
```

означающую «загрузить в регистр 3 данные из ячейки памяти 54», намного легче прочитать, чем эквивалентную последовательность битов. Трудно поверить, но термин «автоматическое программирование» первоначально относился к ассемблерам, так как они автоматически выбирали правильную последовательность битов для каждого символа. Известные языки программирования, такие как C и Pascal, сложнее ассемблерных языков, потому что они «автоматически» выбирают адреса и регистры и даже «автоматически» выбирают последовательности команд для организации циклов и вычисления арифметических выражений.

Теперь мы готовы ответить на вопрос из названия этой главы.

Язык программирования — это механизм абстрагирования. Он дает возможность программисту описать вычисления абстрактно и в то же время позволяет программе (обычно называемой ассемблером, компилятором или интерпретатором) перевести это описание в детализированную форму, необходимую для выполнения на компьютере.

Теперь понятно, почему существуют сотни языков программирования: для двух разных классов задач скорее всего потребуются различные уровни абстракции, и у разных программистов будут различные представления о том, какими должны быть эти абстракции. Программист, работающий на C, вполне доволен работой на уровне абстракции, требующем определения вычислений с помощью массивов и индексов, в то время как составитель отчета отдает предпочтение «программе» на языке, содержащем функции текстовой обработки.

Уровни абстракции легко различить в компьютерных аппаратных средствах. Первоначально монтажные соединения непосредственно связывали дискретные компоненты, такие как транзисторы и резисторы. Затем стали использоваться стандартные подсоединяемые с помощью разъемов модули, за которыми последовали небольшие интегральные схемы. Сегодня компьютеры целиком собираются из горстки чипов, каждый из которых содержит сотни тысяч компонентов. Никакой компьютерщик не рискнул бы разрабатывать «оптимальную» схему из индивидуальных компонентов, если существует набор подходящих чипов, которые выполняют нужные функции.

Из концепции абстракции вытекает общее правило:

Чем выше уровень абстракции, тем больше деталей исчезает.

Если вы пишете программу на C, то теряете возможность задать распределение регистров, которая есть в языке ассемблера; если вы пишете на языке Prolog, то теряете имеющуюся в C возможность определить произвольные связанные структуры с помощью указателей. Существует естественное противоречие между стремлением к краткому, ясному и надежному выражению вычисления на высокоабстрактном уровне и стремлением к гибкости подробного описания вычисления. Абстракция никогда не может быть такой же точной или оптимальной, как описание низкого уровня.

В этом учебнике вы изучите языки трех уровней абстракции. Опуская ассемблер, мы начнем с «обычных» языков программирования, таких как Fortran, C, Pascal и Pascal-подобные конструкции языка Ada. Затем в части 4 мы обсудим языки типа Ada и C ++, которые позволяют программисту создавать абстракции более высокого уровня из операторов обычных языков. В заключение мы опишем языки функционального и логического программирования, работающие на еще более высоком уровне абстракций.

1.2. Процедурные языки

Fortran

Первым языком программирования, который значительно превзошел уровень языка ассемблера, стал Fortran. Он был разработан в 1950-х годах группой специалистов фирмы IBM во главе с Джоном Бекусом и предназначался для абстрактного описания научных вычислений. Fortran встретил сильное противодействие по тем же причинам, что и все последующие предложения абстракций более высокого уровня, а именно из-за того, что большинство программистов полагало, что сгенерированный компилятором программный код не может быть лучше написанного вручную на языке ассемблера.

Подобно большинству первых языков программирования, Fortran имел серьезные недостатки в деталях самого языка, и, что важнее, в нем отсутствовала поддержка современных концепций структурирования модулей и данных. Сам Бекус, оглядываясь назад, говорил:

Мы просто придумывали язык по мере его осмысления. Мы расценивали проектирование языка не как трудную задачу, а просто как прелюдию к реальной проблеме: проектированию компилятора, который мог бы генерировать эффективные программы.

Однако преимущества абстракции быстро покорили большинство программистов: разработка программ стала более быстрой и надежной, а их машинная зависимость уменьшилась из-за абстрагирования от регистров и машинных команд. Поскольку самыми первыми на компьютерах рассчитывались научные задачи, Fortran стал стандартным языком в науке и технике, и только теперь на смену ему приходят другие языки. Fortran был неоднократно модернизирован (1966,1977,1990) с тем, чтобы адаптировать его к требованиям современных программных разработок.

Cobol и PL/1

Язык Cobol был разработан в 1950-х для обработки коммерческих данных. Он создавался комитетом, состоящим из представителей Министерства Обороны США, производителей компьютеров и коммерческих организаций типа страховых компаний. Предполагалось, что Cobol — это только временное решение, необходимое, пока не создан лучший проект; однако язык быстро стал самым распространенным в своей области (как Fortran в науке), причем по той же самой причине: он обеспечивал естественные средства выражения вычислений, типичных для своей области. При обработке коммерческих данных необходимо делать относительно простые вычисления для большого числа сложных записей данных, а

по возможностям структурирования данных Cobol намного превосходит алгоритмические языки типа Fortran или C.

IBM позже создала язык PL/1, универсальный, обладающий всеми свойствами языков Fortran, Cobol и Algol. PL/1 заменил Fortran и Cobol на многих компьютерах IBM, но этот язык очень широкого диапазона никогда не поддерживался вне IBM, особенно на мини- и микроЭВМ, которые все больше и больше используются в организациях, занимающихся обработкой данных.

Algol и его потомки

Из ранних языков программирования Algol больше других повлиял на создание языков. Разработанный международной группой первоначально для общих и научных приложений, он никогда не достигал такой популярности, как Fortran, поскольку не имел той поддержки, которую Fortran получил от большинства производителей компьютеров. Описание первой версии языка Algol было опубликовано в 1958 г.; пересмотренная версия, Algol 60, широко использовалась в компьютерных научных исследованиях и была реализована на многих машинах, особенно в Европе. Третья версия языка, Algol 68, пользовалась влиянием среди теоретиков по языкам, хотя никогда широко не применялась.

От языка Algol произошли два важных языка: Jovial, который использовался Военно-воздушными силами США для систем реального времени, и Simula, один из первых языков моделирования. Но, возможно, наиболее известным его потомком является Pascal, разработанный в конце 1960-х Никлаусом Виртом. Целью разработки было создание языка, который можно было бы использовать для демонстрации идей объявления типов и контроля их соответствия. В последующих главах мы докажем, что эти концепции относятся к наиболее важным, когда-либо предлагавшимся в проектировании языков.

Как язык практического программирования Pascal имеет одно большое преимущество и один большой недостаток. Первоначальный компилятор языка Pascal был написан на самом языке Pascal и, таким образом, мог быть легко перенесен на любой компьютер. Язык распространялся быстро, особенно на создаваемых в то время мини- и микроЭВМ. К сожалению, как язык, Pascal слишком мал. Стандартный Pascal вообще не имеет никаких средств для деления программы на модули, хранящиеся в отдельных файлах, и поэтому не может использоваться для программ объемом больше нескольких тысяч строк. Компиляторы Pascal, используемые на практике, поддерживают декомпозицию на модули, но никаких стандартных методов для этого не существует, так что большие программы непереносимы. Вирт сразу понял, что модули являются необходимой частью любого практического языка, и разработал язык Modula. Modula (теперь в версии 3, поддерживающей объектно-ориентированное программирование) — популярная альтернатива нестандартным «диалектам» языка Pascal.

C

Язык C был разработан в начале 1970-х Деннисом Ричи, сотрудником Bell Laboratories, как язык реализации операционной системы UNIX. Операционные системы традиционно писали на ассемблере, поскольку языки высокого уровня считались неэффективными. Язык C абстрагируется от деталей программирования, присущих ассемблерам, предлагая структурированные управляющие операторы и структуры данных (массивы и записи) и сохраняя при этом всю гибкость ассемблерного низкоуровневого программирования (указатели и операции на уровне битов).

Так как система UNIX была легко доступна для университетов и написана на переносимом языке, а не на языке ассемблера, то она быстро стала популярна в академических и исследовательских учреждениях. Когда новые компьютеры и прикладные программы выходили из этих учреждений на коммерческий рынок, вместе с ними распространялись UNIX и C.

Язык C проектировался так, чтобы быть близким к языку ассемблера, и это обеспечивает ему чрезвычайную гибкость; но проблема состоит в том, что эта гибкость обусловливает чрезвычайную легкость создания программ со скрытыми ошибками, поскольку ненадежные конструкции не проверяются компилятором, как это делается на языке Pascal. Язык C — тонкий инструмент в руках профессионала и удобен для небольших программ, но при разработке на нем больших программных систем группами разработчиков разной квалификации могут возникнуть серьезные проблемы. Мы отметим многие опасные конструкции C и укажем, как не попадать в главные ловушки.

Язык C был стандартизирован в 1989 г. Американским Национальным Институтом Стандартов (ANSI); практически тот же самый стандарт был принят Международной Организацией по Стандартизации (ISO) годом позже. В этой книге делаются ссылки на ANSI C, а не на более ранние версии языка.

C++

В 1980-х годах Бьярн Строуструп, также из Bell Laboratories, использовал C как базис языка C++, добавив поддержку объектно-ориентированного программирования, аналогичную той, которую предоставлял язык Simula. Кроме того, в C++ исправлены многие ошибки языка C, и ему следует отдавать предпочтение даже в небольших программах, где объектно-ориентированные свойства, возможно, и не нужны. C++ — наиболее подходящий язык для обновления систем, написанных на C.

Обратите внимание, что C++ — развивающийся язык, и в вашем справочном руководстве или компиляторе, возможно, отсутствуют последние изменения. Обсуждаемый в этой книге язык соответствует книге *Annotated C++ Reference* Manual Эллиса и Строуструпа (издание 1994 г.), которая является основой рассматриваемого в настоящее время стандарта.

Ada

В 1977 г. Министерство Обороны Соединенных Штатов решило провести унификацию языка программирования, в основном, чтобы сэкономить на обучении и стоимости поддержки операционных сред разработки программ для различных военных систем. После оценки существующих языков было принято решение провести конкурс на разработку нового языка, положив в основу хороший существующий язык, такой как Pascal. В конце концов был выбран язык, который назвали Ada, и стандарт был принят в 1983 г. Язык Ada уникален в ряде аспектов:

• Большинство языков (Fortran, C, Pascal) создавались едиными командами разработчиков и были стандартизованы уже после их широкого распространения. Для сохранения совместимости все случайные промахи исходной команды включались в стандарт. Ada же перед стандартизацией подверглась интенсивной проверке и критическому разбору.

• Многие языки первоначально были реализованы на единственном компьютере, и на них сильно повлияли особенности этого компьютера. Язык Ada был разработан для написания переносимых программ.

• Ada расширяет область применения языков программирования, обеспечивая обработку ошибок и параллельное программирование, что традиционно считалось (нестандартными) функциями операционных систем.

Несмотря на техническое совершенство и преимущества ранней стандартизации, язык Ada не достиг большой популярности вне военных и других крупномасштабных проектов (типа коммерческой авиации и железнодорожных перевозок). Язык Ada получил репутацию

трудного. Это связано с тем, что он поддерживает многие аспекты программирования (параллелизм, обработку исключительных ситуаций, переносимые числовые данные), которые другие языки (подобные C и Pascal) оставляют операционной системе. На самом деле его нужно просто больше изучать. К тому же первоначально были недоступны хорошие и недорогие среды разработки для сферы образования. Теперь, когда есть бесплатные компиляторы (см. приложение A) и хорошие учебники, Ada все чаще и чаще встречается в учебных курсах даже как «первый» язык.

Ada 95

Ровно через двенадцать лет после принятия в 1983 г. первого стандарта языка Ada был издан новый стандарт. В новой версии, названной Ada 95, исправлены некоторые ошибки первоначальной версии, но главное — это добавление поддержки настоящего объектно-ориентированного программирования, включая наследование, которого не было в Ada 83, так как его считали неэффективным. Кроме того, Ada 95 содержит приложения, в которых описываются стандартные (но необязательные) расширения для систем реального времени, распределенных систем, информационных систем, защищенных систем, а также «числовое» (numerics) приложение.

В этой книге название «Ada» будет использоваться в тех случаях, когда обсуждение не касается особенностей одной из версий: Ada 83 или Ada 95. Заметим, что в литературе Ada 95 упоминалась как Ada 9X, так как во время разработки точная дата стандартизации не была известна

.

1.3. Языки, ориентированные на данные

На заре программирования было создано и реализовано несколько языков, значительно повлиявших на дальнейшие разработки. У них была одна общая черта: каждый язык имел предпочтительную структуру данных и обширный набор команд для нее. Эти языки позволяли создавать сложные программы, которые трудно было бы написать на языках типа Fortran, просто манипулирующих компьютерными словами. В следующих подразделах мы рассмотрим некоторые из этих языков.

Lisp

Основная структура данных в языке Lisp — связанный список. Первоначаль-но Lisp был разработан для исследований в теории вычислений, и многие работы по искусственному интеллекту были выполнены на языке Lisp. Язык был настолько важен, что компьютеры разрабатывались и создавались так, чтобы оптимизировать выполнение Lisp-программ. Одна из проблем языка состояла в обилии различных «диалектов», возникавших по мере того, как язык реализовывался на различных машинах. Позже был разработан стандартный язык Lisp, чтобы программы можно было переносить с одного компьютера на другой. В настоящее время популярен «диалект» языка Lisp — CLOS, поддерживающий объектно-ориентированное программирование.

Три основные команды языка Lisp — это car(L) и cdr(L), которые извлекают, соответственно, начало и конец списка L, и cons(E, L), которая создает новый список из элемента E и существующего списка L. Используя эти команды, можно определить функции обработки списков, содержащих нечисловые данные; такие функции было бы довольно трудно запрограммировать на языке Fortran.

Мы не будем больше обсуждать язык Lisp, потому что многие его основополагающие идеи были перенесены в современные функциональные языки программирования типа ML, который мы обсудим в гл. 16.

APL

Язык APL является развитием математического формализма, который используется для описания вычислений. Основные структуры данных в нем — векторы и матрицы, и операции выполняются над такими структурами непосредственно, без циклов. Программы на языке APL очень короткие по сравнению с аналогичными программами на традиционных языках. Применение APL осложняло то, что в язык перешел большой набор математических символов из первоначального формализма. Это требовало специальных терминалов и затрудняло экспериментирование с APL без дорогостоящих аппаратных средств. Современные графические интерфейсы пользователя, применяющие программные шрифты, решили эту проблему, которая замедляла принятие APL.

Предположим, что задана векторная переменная:

V= 1 5 10 15 20 25

Операторы языка APL могут работать непосредственно с V без записи циклов с индексами:

```
+ /V      =76 Свертка сложением(суммирует элементы)
 ⌽V       =   25 20 15 10 5 1          Обращает вектор

2 3 ⍴V  =        1     5    10 Переопределяет размерность
              V 15    20    25          как матрицу 2x3
```

Векторные и матричные сложения и умножения также можно выполнить непосредственно над такими переменными.

Snobol, Icon

Первые языки имели дело практически только с числами. Для работы в таких областях, как обработка естественных языков, идеально подходит Snobol (и его преемник Icon), поскольку их базовой структурой данных является строка. Основная операция в языке Snobol сравнивает образец со строкой, и побочным результатом совпадения может быть разложение строки на подстроки. В языке Icon основная операция — вычисление выражения, причем выражения включают сложные операции со строками.

В языке Icon есть важная встроенная функция find(s1, s2), которая ищет вхождения строки s1 в строку s2. В отличие от подобных функций языка C find *генерирует* список всех позиций в s2, в которых встречается s1:

```
line := 0                                   # Инициализировать счетчик строк
while s := read() {                          # Читать до конца файла
        every col := find("the", s) do       # Генерировать позиции столбца
write (line, " ",col)                        # Write(line,col) для "the"
line := line+ 1
}
```

Эта программа записывает номера строк и столбцов всех вхождений строки "the" в файл. Если команда find не находит ни одного вхождения, то она «терпит неудачу» (fail), и вычисление выражения завершается. Ключевое слово every вызывает повторение вычисления функции до тех пор, пока оно завершается успешно.

Выражения Icon содержат не только строки, которые представляют собой последовательности символов; они также определены над наборами символов csets. Таким образом

```
vowels := 'aeiou'
```

присваивает переменной vowel (гласные) значение, представляющее собой набор указанных символов. Это можно использовать в функциях типа upto(vowels,s), генерирующих последовательность позиций гласных в s, и many(vowels,s), возвращающих самую длинную начальную последовательность гласных в s.

Более сложная функция bal подобна upto за исключением того, что она генерирует последовательности позиций, которые сбалансированы по «скобочным» символам:

```
bal(' +-*/','([', ')]',*)
```

Это выражение могло использоваться в компиляторе, чтобы генерировать сбалансированные арифметические подстроки. Если в качестве строки s задать "x + (y [u/v] - 1)*z", вышеупомянутое выражение сгенерирует индексы, соответствующие подстрокам:

```
x

x+(y[u/v]-1
```

Первая подстрока сбалансирована, так как она заканчивается «+» и не содержит никаких скобок; вторая подстрока сбалансирована, поскольку она завершается символом «*» и имеет квадратные скобки, правильно вложенные внутри круглых скобок.

Так как вычисление выражения может быть неуспешным (fail), используется *откат (backtracking),* чтобы продолжить поиск от предыдущих генерирующих функций. Следующая программа печатает вхождения гласных, за исключением тех, которые начинаются в столбце 1 .

```
line := 0                                   # Инициализировать счетчик строк
while s := read() {                         # Читать до конца файла
every col := (upto (vowels, line) > 1 ) do
                                            # Генерировать позиции столбца
write (line, " ",col)                       # write(line,col) для гласных
 line := line + 1
}
```

Функция поиска генерирует индекс, который затем проверяется на «>». Если проверка неуспешна (не говорите: «если результат ложный»), программа возвращает управление генерирующей функции upto, чтобы получить новый индекс.

Icon — удобный язык для программ, выполняющих сложную обработку строк. В нем происходит абстрагирование от большей части явных индексных вычислений, и программы оказываются очень короткими по сравнению с обычными языками для числового или системного программирования. Кроме того, в Icon очень интересен встроенный механизм генерации и отката, который предлагает более развитый уровень абстракции управления.

SETL

Основная структура данных в SETL — множество. Так как множество — наиболее общая математическая структура, с помощью которой определяются все другие математические структуры, то SETL может использоваться для создания программ высокой степени общности и абстрактности и поэтому очень коротких. Такие программы имеют сходство с логическими программами (гл. 17), в которых математические описания могут быть непосредственно исполняемыми. В теории множеств используется нотация: *{x \p(x)}*, обозначающая множество всех *x* таких, что логическое выражение *p(x)* является истиной. Например, множество простых чисел в этой нотации может быть записано как

{ *n* \ -,3 *m [(2<m<n*— 1) л *(nmodm* = 0)]}

Эта формула читается так: множество натуральных чисел *n* таких, что не существует натурального *m* от 2 до *n* — 1 , на которое *n* делится без остатка.

Чтобы напечатать все простые числа в диапазоне от 2 до 100, достаточно «протранслировать» это определение в однострочную программу на языке SETL:

```
print ({n in {2.. 100} | not exists m in {2.. n — 1} | (n mod m) = 0});
```

Все эти языки объединяет то, что к их разработке подходили больше с математических, чем с инженерных позиций. То есть решалась задача, как смоделировать известную математическую теорию, а не как выдавать команды для процессора и памяти. Такие продвинутые языки очень полезны для трудно программируемых задач, где важно сосредоточить внимание на проблеме, а не на деталях реализации.

Языки, ориентированные на данные, сейчас несколько менее популярны, чем раньше, отчасти потому, что объектно-ориентированные методы позволяют внедрить операции, ориентированные на данные, в обычные языки типа C++ и Ada, а также из-за конкуренции более новых языковых концепций, таких как функциональное и логическое программирование. Тем не менее эти языки интересны с технической точки зрения и вполне подходят для решения задач, для которых они были разработаны. Студентам рекомендуется приложить усилие и изучить один или несколько таких языков, потому что это расширит их представление о том, как может быть структурирован язык программирования.

1.4. Объектно-ориентированные языки

Объектно-ориентированное программирование (ООП) — это метод структурирования программ путем идентификации объектов реального мира или других объектов и написания модулей, каждый из которых содержит все данные и исполняемые команды, необходимые для представления одного *класса* объектов. Внутри такого модуля существует отчетливое различие между абстрактными свойствами класса, которые экспортируются для использования другими объектами, и реализацией, которая скрыта таким образом, что может быть изменена без влияния на остальную часть системы.

Первый объектно-ориентированный язык программирования Simula был создан в 1960-х годах К. Нигаардом и О.-Дж. Далом для моделирования систем: каждая подсистема, принимающая участие в моделировании, программировалась как объект. Так как возможно существование нескольких экземпляров одной подсистемы, то можно запрограммировать класс для описания каждой подсистемы и выделить память для объектов этого класса.

Исследовательский центр Xerox Palo Alto Research Center популяризировал ООП с помощью языка Smalltalk. Такие же исследования вели к системам окон, так популярным

сегодня, но важное преимущество Smalltalk заключается в том, что это не только язык, но и полная среда программирования. В техническом плане Smalltalk был достижением как язык, в котором классы и объекты являются *единственными* конструкциями структурирования, так что нет необходимости встраивать эти понятия в «обычный» язык.

Технический аспект этих первых объектно-ориентированных языков помешал более широкому распространению ООП: отведение памяти, диспетчеризация операций и контроль соответствия типов осуществлялись динамически (во время выполнения), а не статически (во время компиляции). Не вдаваясь в детали (см. соответствующий материал в гл. 8 и 14), отметим, что в итоге программы на этих языках связаны с неприемлемыми для многих систем накладными расходами по времени и памяти. Кроме того, статический контроль соответствия типов (см. гл. 4) теперь считается необходимым для разработки надежного программного обеспечения. По этим причинам в языке Ada 83 реализована только частичная поддержка ООП.

Язык C++ показал, что можно реализовать полный механизм ООП способом, который совместим со *статическим* распределением памяти и контролем соответствия типов и с фиксированными затратами на диспетчеризацию: динамические механизмы ООП используются только, если они необходимы до существу. Поддержка ООП в Ada 95 основана на тех же идеях, что и в C++.

Однако нет необходимости «прививать» поддержку ООП в существующие языки, чтобы получить эти преимущества. Язык Eiffel подобен Smalltalk в том, что единственным методом структурирования является метод классов и объектов, а также подобен C++ и Ada 95 в том, что проверка типов статическая, а реализация объектов может быть как статической, так и динамической, если нужно. Простота языка Eiffel по сравнению с гибридами, которым «привита» полная поддержка ООП, делает его превосходным выбором в качестве первого языка программирования.

Мы обсудим языковую поддержку ООП более подробно, сначала в C++, а затем в Ada 95. Кроме того, краткое описание Eiffel покажет, как выглядит «чистый» язык ООП.

1.5. Непроцедурные языки

Все языки, которые мы обсудили, имеют одну общую черту: базовый оператор в них — это оператор присваивания, который заставляет компьютер переместить данные из одного места в другое. В действительности это относительно низкий уровень абстракции по сравнению с уровнем проблем, которые мы хотим решать с помощью компьютера. Более новые языки скорее предназначены для того, чтобы описывать проблему и перекладывать на компьютер выяснение, как ее решить, чем для подробного определения, как перемещать данные.

Современные программные пакеты (software packages), как правило, представляют собой языки действительно высокого уровня абстракции. Генератор I Приложений позволяет вам *описать* последовательность экранов и структур базы данных и по этим описаниям автоматически генерирует команды, реализующие ваше приложение. Точно также электронные таблицы, настольные издательские системы, пакеты моделирования и другие системы имеют обширные средства абстрактного программирования. Недостаток программного обеспечения этого типа в том, что оно обычно ограничивается приложениями, которые можно легко запрограммировать. Их можно назвать *параметризованными программами* в том смысле, что, получая описания как параметры, пакет конфигурирует себя для выполнения нужной вам программы.

Другой подход к абстрактному программированию состоит в том, чтобы описывать вычисление, используя уравнения, функции, логические импликации или другие формализмы подобного рода. Благодаря математическим формализмам определенные таким образом языки оказываются действительно универсальными, выходящими за рамки конкретных прикладных областей. Компилятор реально не преобразует программу в машинные коды; скорее, он пытается решать математическую проблему и ее решение

выдает в качестве результата. Так как абстракция оставляет индексы, указатели, циклы и т. п. вне языка, эти программы могут быть на порядок короче обычных программ. Основная проблема описательного программирования состоит в том, что «процедурные» задачи, например ввод-вывод на экран или диск, плохо «укладываются» в эту концепцию, и для этих целей языки должны быть дополнены обычными конструкциями программирования.

Мы обсудим два формализма непроцедурных языков: 1) функциональное программирование (гл. 16), которое основано на математическом понятии чистой функции, такой как sin и log, которые не изменяют своего окружения в отличие от так называемых функций обычного языка типа C, которые могут иметь побочные эффекты; 2) логическое программирование (гл. 17), в котором программы выражены как формулы математической логики и «компилятор», чтобы решить задачу, пытается вывести логические следствия из этих формул.

Должно быть очевидно, что программы на абстрактных, непроцедурных языках не могут оказаться столь же эффективными, как программы, закодированные вручную на C. Непроцедурным языкам следует отдавать предпочтение всякий раз, когда программная система должна осуществлять поиск в больших объемах информации или решать задачи, процесс решения которых не может быть точно описан. Примерами могут служить обработка текстов (перевод, проверка стиля), распознавание образов (видение, генетика) и оптимизация процессов (планирование). Поскольку методы реализации улучшаются и поскольку становится все сложнее разрабатывать надежные программные системы на обычных языках, области приложений непроцедурных языков будут расширяться.

Функциональные и логические языки программирования настоятельно рекомендуются как первые из изучаемых, для того чтобы студенты с самого начала учились работать на более высоких уровнях абстракции, чем при программировании на таких языках, как Pascal или C.

1.6. Стандартизация

Следует подчеркнуть значение стандартизации. *Если* для языка существует стандарт, и *если* компиляторы его поддерживают, то программы можно переносить с одного компьютера на другой. Когда вы пишете пакет программ, который должен выполняться на разных компьютерах, вы должны строго придерживаться стандарта. Иначе задача сопровождения чрезвычайно усложнится, потому что придется следить за десятками или сотнями машинно-зависимых вопросов.

Стандарты существуют (или находятся в стадии подготовки) для большинства языков, обсуждаемых в этой книге. К сожалению, обычно стандарт предлагается спустя годы после широкого распространения языка и должен сохранять машинно-зависимые странности ранних реализаций. Язык Ada — исключение в том смысле, что стандарты (1983 и 1995) создавались и оценивались одновременно с проектом языка и первоначальной реализацией. Более того, стандарт ориентирован на то, чтобы компиляторы можно было сравнивать по производительности и стоимости, а не только на соответствие стандарту. Компиляторы зачастую могут предупреждать вас, если вы использовали нестандартную конструкцию. Если необходимо использовать такие конструкции, их следует сконцентрировать в нескольких хорошо документированных модулях.

1.7. Архитектура компьютера

Поскольку мы рассматриваем языки программирования с точки зрения их практического использования, мы включаем короткий раздел по архитектуре компьютеров, чтобы согласовать минимальный набор терминов. Компьютер состоит из *центрального процессора* (ЦП) и *памяти* (рис. 1.1). Устройства ввода-вывода могут рассматриваться как частный случай памяти.

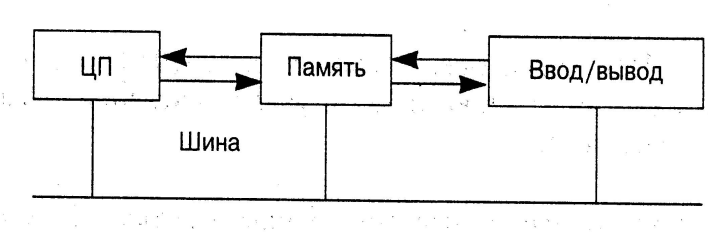

рис. 1 . 1 . Архитектура компьютера.

Все компоненты компьютера обычно подсоединяются к общей *шине.* Физически шина — это набор разъемов, соединенных параллельно; логически шина — это спецификация сигналов, которые дают возможность компонентам обмениваться данными. Как показано на рисунке, современные компьютеры могут иметь дополнительные прямые соединения между компонентами для повышения производительности (путем специализации интерфейса и расширения узких мест). С точки зрения программного обеспечения единственное различие состоит в скорости, с которой данные могут передаваться Между компонентами.

В ЦП находится *набор регистров* (специальных ячеек памяти), в которых выполняется вычисление. ЦП может выполнить любую хранящуюся в памяти *команду;* в ЦП есть *указатель команды,* который указывает на расположение очередной команды, которая будет выполняться. Команды разделены на Следующие классы.

• Доступ к памяти. Загрузить (load) содержимое слова памяти в регистр и сохранить (store) содержимое регистра в слове памяти.

• Арифметические команды типа сложить (add) и вычесть (sub). Эти действия выполняются над содержимым двух регистров (или иногда над содержимым регистра и содержимым слова памяти). Результат остается в регистре. Например, команда

```
add m,N   R1,N
```

складывает содержимое слова памяти N с содержимым регистра R1 и оставляет результат в регистре.

• Сравнить и перейти. ЦП может сравнивать два значения, такие как содержимое двух регистров; в зависимости от результата (равно, больше, и т.д.) указатель команды изменяется, переходя к другой команде. Например:

```
        jump_eq    R1.L1
          …
L1:     ...
```

заставляет ЦП продолжать вычисление с команды с меткой L1, если со-держимое R1 — ноль; в противном случае вычисление продолжается со следующей команды.

Во многих компьютерах, называемых *Компьютерами с Сокращенной Системой команд команд* (RISC— Reduced Instruction Set Computers), имеются только такие элементарные команды.Обосновывается это тем, что ЦП, который должен выполнять всего несколько простых команд, может быть очень быстрым. В других компьютерах, известных как CISC (Complex Instruction Set Computers), определен *Сложный Набор команд*, позволяющий упростить как программирование на языке ассемблера, так и конструкцию компилятора. Обсуждение этих двух подходов выходит за рамки этой книги; у них достаточно много общего, так что выбор не будет иметь для нас существенного значения.

Память — это набор ячеек, в которых можно хранить данные. Каждая ячейка памяти, называемая *словом памяти*, имеет *адрес*, а каждое слово состоит из фиксированного числа битов, обычно из 16, 32 или 64 битов. Возможно, что Компьютер умеет загружать и сохранять 8-битовые *байты* или *двойные слова* из 64 битов.

Важно знать, какие способы адресации могут использоваться в команде. Самый простой способ — непосредственная адресация, при которой операндявляется частью команды. Значением операнда может быть адрес переменной, и в этом случае мы используем нотацию C:

```
load R3, # 54                    Загрузить значение 54 в R3 load
 R2, &N                          Загрузить адрес N в R2
```

Следующий способ — это абсолютная адресация, в которой обычно используется символический адрес переменной:

```
load    R3,54      Загрузить содержимое адреса 54
load    R4, N      Загрузить содержимое переменной N
```

Современные компьютеры широко используют *индексные регистры*. Индексные регистры не обязательно обособлены от регистров, используемых для вычислений; важно, что содержимое индексного регистра может использоваться для вычисления адреса операнда команды. Например:

```
load    R3,54(R2)     Загрузить содержимое addr(R2) + 54
load    R4, (R1)      Загрузить содержимое addr(R1) + O
```

где первая команда означает «загрузить в регистр R3 содержимое слова памяти, чей адрес получен, добавлением 54 к содержимому (индексного) регистра R2»; вторая команда — это частный случай, когда содержимое регистра R1 используется просто как адрес слова памяти, содержимого которого загружается в R4. Индексные регистры необходимы для эффективной реализации циклов и массивов.

Кэш и виртуальная память

Одна из самых трудных проблем, стоящих перед архитекторами компьютеров, — это приведение в соответствие производительности ЦП и пропускной способности памяти. Быстродействие ЦП настолько велико по сравнению со временем доступа к памяти, что память не успевает поставлять данные, чтобы обеспечить непрерывную работу процессора. Для этого есть две причины: 1) в компьютере всего несколько процессоров (обычно один), и в них можно использовать самую быструю, наиболее дорогую технологию, но объем памяти постоянно наращивается и технология должна быть менее дорогая; 2) скорости настолько высоки, что ограничивающим фактором является быстрота, с которой электрический сигнал распространяется по проводам между ЦП и памятью.

Решением проблемы является использование иерархии блоков памяти, как показано на рис. 1.2. Идея состоит в том, чтобы хранить неограниченное количество команд программы и данных в относительно медленной (и недорогой) памяти и загружать порции необходимых команд и данных в меньший объем быстрой (и дорогой) памяти. Если в качестве медленной памяти ис пользуется диск, а в качестве быстрой памяти — обычная оперативная память с произвольным

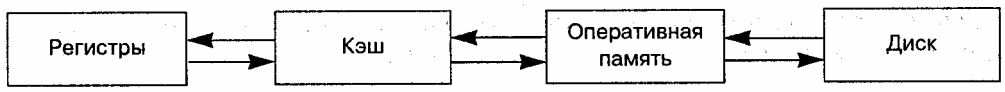

Рис. 1.2. Кэш и виртуальная память.

доступом (RAM — Random Access Memory), то концепция называется *виртуальной памятью* или *страничной памятью*. Если медленной памятью является RAM, а быстрой — RAM, реализованная по более быстрой технологии, то концепция называется *кэш-памятью*.

Обсуждение этих концепций выходит за рамки этой книги, но программист должен понимать потенциальное воздействие кэша или виртуальной памяти на программу, даже если функционирование этих блоков памяти обеспечивается компьютерными аппаратными средствами или операционной системой и полностью невидимо для программиста. Команды и данные передаются между медленной и быстрой памятью блоками, а не отдельными словами. Это означает, что исполнение последовательно расположенных команд без переходов, так же как и обработка, последовательно расположенных данных (например, просмотр элементов массива), должны быть намного эффективнее, чем исполнение групп команд с переходами и обращения к памяти в случайном порядке, что требует интенсивного обмена блоками информации между различными иерархическими уровнями памяти. Если вы пытаетесь улучшать эффективность программы, то следует противиться искушению писать куски на языках низшего уровня или ассемблере; вместо этого попытайтесь реорганизовать вычисление, приняв во внимание влияние кэша и виртуальной памяти. Перестановка операторов языка высокого уровня не воздействует на переносимость программы, хотя, конечно, улучшение эффективности может теряться при перенесении ее на компьютер с иной архитектурой.

1.8. Вычислимость

В 1930-х годах, еще до того, как были изобретены компьютеры, логики исследовали абстрактные концепции вычисления. Алан Тьюринг и .Алонзо Черч независимо предложили чрезвычайно простые модели вычисления (названные соответственно *машинами Тьюринга* и *Лямбда-исчислением) и* затем выдвинули следующее утверждение (известное как *Тезис Черча —Тьюринга):*

Любое исполнимое вычисление может быть выполнено на любой из этих моделей.

Машины Тьюринга чрезвычайно просты; если воспользоваться синтаксисом языка С, то объявления данных будут выглядеть так:

```
char tape[...];
int current = 0;
```

где лента (tape) предполагается бесконечной. Программа состоит из любого числа операторов вида:

```
L17:    if (tape[currentj == 'g') {
            tape[current++] = 'j'i
            goto L43;
        }
```

Оператор машины Тьюринга выполняется за четыре следующих шага.

• Считать и проверить символ в текущей ячейке ленты.

• Заменить символ на другой символ (необязательно).

• Увеличить или уменьшить указатель текущей ячейки.

• Перейти к другому оператору.

Согласно Тезису Черча — Тьюринга, любое вычисление, которое действительно можно описать, может быть запрограммировано на этой примитивной машине. Интуитивная очевидность Тезиса опирается на два утверждения:

• Исследователи предложили множество моделей вычислений, и было доказано, что все они эквивалентны машинам Тьюринга.

• Никому пока не удалось описать вычисление, которое не может быть реализовано машиной Тьюринга.

Так как машину Тьюринга можно легко смоделировать на любом языке программирования, можно сказать, что все языки программирования «делают» одно и то же, т. е. в некотором смысле эквивалентны.

1.9. Упражнения

1. Опишите, как реализовать компилятор для языка на том же самом языке («самораскрутка»).

2. Придумайте синтаксис для APL-подобного языка для матричных вычислений, используя обычные символы.

3. Составьте список полезных команд над строками и сравните ваш список со встроенными командами языков Snobol и Icon.

4. Составьте список полезных команд над множествами и сравните ваш список со встроенными командами языка SETL.

5. Смоделируйте (универсальную) машину Тьюринга на нескольких языках программирования.

Глава 2

Элементы языков программирования

2.1. Синтаксис

Как и у обычных языков, у языков программирования есть синтаксис:

Синтаксис языка (программирования) — это набор правил, которые определяют, какие последовательности символов считаются допустимыми выражениями (программами) в языке.

Синтаксис задается с помощью формальной нотации.

Самая распространенная формальная нотация синтаксиса — *это расширенная форма Бекуса — Наура (РБНФ).* В РБНФ мы начинаем с объекта самого верхнего уровня, *с программы,* и применяем правила декомпозиции объектов, пока не достигнем уровня отдельного символа. Например, в языке С синтаксис условного оператора (if-оператора) задается правилом:

if-оператор :: = if *(выражение) оператор* [else *оператор*]

Имена, выделенные курсивом, представляют синтаксические категории, а имена и символы, выделенные полужирным шрифтом, представляют фактические символы, которые должны появиться в программе. Каждое правило содержит символ «:: =», означающий «представляет собой». Прочие символы используются для краткости записи:

[ ] Не обязательный {} Ноль или более повторений | Или

Таким образом, else-оператор в if-операторе не является обязательным. Использование фигурных скобок можно продемонстрировать на (упрощенном) правиле для объявления списка переменных:

Объявление-переменной ::= *спецификатор-типа идентификатор {, идентификатор};*

Это читается так: объявление переменной *представляет собой* спецификатор типа, за которым следует идентификатор (имя переменной) и необязательная последовательность идентификаторов, предваряемых запятыми, в конце ставится точка с запятой.

Правила синтаксиса легче изучить, если они заданы в виде диаграмм (рис. 2.1). Круги или овалы обозначают фактические символы, а прямоугольники — синтаксические категории, которые имеют собственные диаграммы.

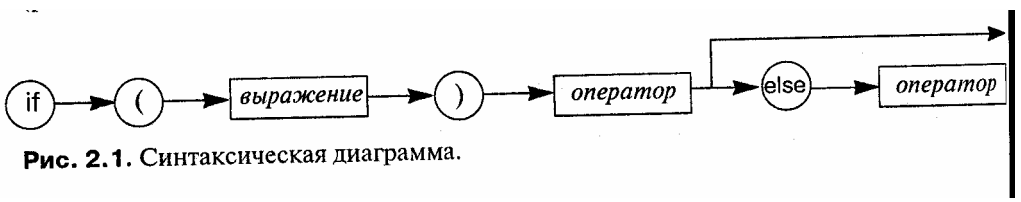

Рис. 2.1. Синтаксическая диаграмма.

.

Последовательность символов, получаемых при последовательном прохождении пути на диаграммах, является (синтаксически) правильной программой.

Хотя многие программисты страстно привязаны к синтаксису определенного языка, этот аспект языка, пожалуй, наименее важен. Любой разумный синтаксис легко изучить; кроме того, синтаксические ошибки обнаруживаются компилятором и редко вызывают проблемы с работающей программой. Мы ограничимся тем, что отметим несколько возможных синтаксических ловушек, которые могут вызвать ошибки во время выполнения программы:

Будьте внимательны с ограничениями на длину идентификаторов. Если значимы только первые 10 символов, то current_winner и current _width будут представлять один и тот же идентификатор.

Многие языки не чувствительны к регистру, то есть СЧЕТ и счет пред-ставляют одно и то же имя. Язык С чувствителен к регистру, поэтому эти имена представляют два разных идентификатора. При разработке чувствительных к регистру языков полезно задать четкие соглашения по использованию каждого регистра, чтобы случайные опечатки не приводили к ошибкам. Например, по одному из соглашений языка С в программе все записывается на нижнем регистре за исключением определенных имен констант, которые задаются на верхнем регистре.

Существуют две формы комментариев: комментарии в языках Fortran, Ada и C++ начинаются с символа (С, - -, и //, соответственно) и распространяются до конца строки, в то время как а языках С и Pascal комментарии имеют как начальный, так и конечный символы: /* ... */ в.С и (* ... *) иди {...} в Pascal. Вторая форма удобна для «закомментаривания» неиспользуемого кода (который, взможно, был вставлен для тестированя), но при этом существует опасность пропустить конечный символ, в результате чего будет пропущена последовательность операторов:

```
/*              Комментарий следовало бы закончить здесь
a = b + c;      Оператор будет пропущен
/*...*/         Здесь конец комментария
```

C

Остерегайтесь похожих, но разных символов. Если вы когда-либо изучали математику, то вас должно удивить, что знакомый символ «=» используется в языках С и Fortran как оператор присваивания, в то время как новые символы «==» в С и «.eq.» в Fortran используются в качестве операции сравнения на равенство. Стремление написать:

```
if(a=b)...
```

C

является настолько общим, что многие компиляторы выдадут предупреждение, хотя оператор имеет допустимое значение.

В качестве исторического прецедента напомним известную проблему с синтаксисом языка Fortran. Большинство языков требует, чтобы слова в программе отделялись одним или несколькими пробелами (или другими *пробельными* символами типа табуляции), однако в

языке Fortran пробельные символы игнорируются. Рассмотрим следующий оператор, который определяет «цикл до метки 10 при изменении индекса i от 1 до 100»:

Fortan

```
do 10 i = 1,100
```

Если запятая случайно заменена точкой, то этот оператор становится на самом деле.оператором присваивания, присваивая 1.100 переменной, имя которой образуется соединением всех символов перед знаком «=»:

Fortan

```
do10i = l.TOO
```

Говорят, эта ошибка заставила ракету взорваться до запуска в космос!

2.2. Семантика

Семантика — это смысл высказывания (программы) в языке (программирования).

Если синтаксис языков понять очень легко, то понять семантику намного труднее. Рассмотрим для примера два предложения на английском языке:

The pig is in the pen. (Свинья в загоне.)
The ink is in the pen. (Чернила в ручке.)

Нужно обладать достаточно обширными общими знаниями, не имеющими никакого отношения к построению английских фраз, чтобы знать, что «реп» имеет разные значения в этих двух предложениях («загон» и «ручка»).

Формализованная нотация семантики языков программирования выходит за рамки этой книги. Мы только кратко изложим основную идею. В любой точке выполнения программы мы можем описать ее *состояние*, определяемое: (1) указателем на следующую команду, которая будет выполнена, и (2) содержимым памяти программы. Семантика команды задается описанием изменения состояния, вызванного выполнением команды. Например, выполнение:

a:=25

заменит состояние s на новое состояние s', которое отличается от s только тем, что ячейка памяти a теперь содержит 25.

Что касается управляющих операторов, то для описания вычисления используется математическая логика. Предположим, что мы уже знаем смыслы двух операторов S1 и S2 в произвольном состоянии s. Обозначим это с помощью формул p (S1, s) и p (S2, s) соответственно. Тогда смысл if-оператора:

if C then S1 elseS2

задается формулой:

(C(s)=>p(S1,s))&((-C(s)=>p(S2,s))

Если вычисление C в состоянии s дает результат *истина*, то смысл if-оператора такой же, как смысл S1; в противном случае вычисление C дает результат *не истина* и смысл if-оператора такой же, как у S2.

Как вы можете себе представить, определение семантики операторов цикла и вызовов процедур с параметрами может быть очень сложным. Здесь мы удовлетворимся неформальными объяснениями семантики этих конструкций языка, как их обычно описывают в справочных руководствах:

Проверяется условие после if; если результат — *истина*, выполняется следующий после then оператор, иначе выполняется оператор, следующий за else.

Формализация семантики языков программирования дает дополнительное преимущество — появляется возможность *доказать* правильность программы. По сути, выполнение программы можно формализовать с помощью Кксиом, которые описывают, как оператор преобразует состояние, удовлетворяющее *утверждению* (логической формуле) на входе, в состояние, которое Удовлетворяет утверждению на выходе. Смысл программы «вычисляется» пу-тем построения входных и выходных утверждений для всей программы на основе утверждений для отдельных операторов. Результатом является доказательство того, что *если* входные данные удовлетворяют утверждению на входе, *то* выходные данные удовлетворяют утверждению на выходе.

Конечно, «доказанную» правильность программы следует понимать лишь относительно утверждений на входе и выходе, поэтому не имеет смысла доказывать, что программа вычисляет квадратный корень, если вам нужна программа для вычисления кубического корня! Тем не менее верификация программы применялась как мощный метод проверки для систем, от которых требуется высокая надежность. Важнее то, что изучение верификации поможет вам писать правильные программы, потому что вы научитесь мыслить, исходя из требований правильности программы. Мы также рекомендуем изучить и использовать язык программирования Eiffel, в который включена поддержка утверждений (см. раздел 11.5).

2.3. Данные

При первом знакомстве & языками программирования появляется тенденция сосредоточивать внимание на действиях: операторах или командах. Только изучив и опробовав операторы языка, мы обращаемся к изучению поддержки, которую обеспечивает язык для структурирования данных. В современных взглядах на программирование, особенно на объектно-ориентированное, операторы считаются средствами манипулирования данными, использующимися для представления некоторого объекта. Таким образом, следует изучить аспекты структурирования данных до изучения действий.

Программы на языке ассемблера можно рассматривать как описания действий, которые должны быть выполнены над *физическими* сущностями, такими как ячейки памяти и регистры. Ранние языки программирования продолжили эту традицию идентифицировать сущности языка, подобные *переменным*, как слова памяти, несмотря на то, что этим переменным приписывались математические атрибуты типа *целое*. В главах 4 и 9 мы объясним, почему int и float — это не математияеское, а скорее, физическое представление памяти.

Теперь мы определим центральную концепцию программирования:

Тип — это множество значений и множество операций над этими значениями.

Правильный смысл int в языке С таков: int — это тип, состоящий из конечного множества значений (количестве примерно 65QOQ или 4 миллиардов, в зависимости от компьютера) и набора операций (обозначенньгх, +, < =, и т.д.) над этими значениями. В таких современных языках программирования, как Ada и C++, есть возможность создавать новые типы. Таким образом, мы больше не ограничены горсткой типов, предопределенных разработчиком языка; вместо этого мы можем создавать собственньхе типы, которые более точно.со-ответствуют решаемой задаче.

При обсуждении типов данных в этой книге используется именно этот подход: определение набора значений и одераций над, этими значениями, Только позднее мы обсудим, как такой тип может быть реализован на копь-ютере. Например, массив — это индексированная совокупность элементов с такими операциями, как индексация, Обратите внимание, что определение типа зависит от языка: операция присваивания над массивами оцределена в языке Ada, но не в языке С. После определения типа массива можно изучать реализацию массивов, как последовательностей ячеек памяти.

В заключение этого раздела мы определим следующие термины, которые будут использоваться при обсуждении данных:

Значение. Простейшее неопределяемое понятие.

Литерал. Конкретное значение, заданное в программе «буквально», в виде последовательности символов, например 154, 45.6, FALSE, 'x', "Hello world".

Представление. Значение, представленное внутри компьютера конкретной строкой битов. Например, символьное значение, обозначенное 'x', может представляться строкой из восьми битов 01111000.

Переменная. Имя, присвоенное ячейке памяти или ячейкам, которые могут содержать представление значения конкретного типа. Значение может изменяться во время выполнения программы.

Константа. Константа является именем, присвоенным ячейке памяти или ячейкам, которые могут содержать представление значения конкретного типа. Значение *не может быть изменено* во время выполнения программы.

Объект. Объект — это переменная или константа.

Обратите внимание, что для переменной должен быть определен конкретный тип по той простой причине, что компилятор должен знать, сколько памяти для нее нужно выделить! Константа — это просто переменная, которая не может изменяться. Пока мы не дошли до обсуждения объектно-ориентированного программирования, мы будем использовать знакомый термин «переменная» в более общем смысле, для обозначения и константы, и переменной вместо точного термина «объект».

2.4. Оператор присваивания

Удивительно, что в обычных языках программирования есть только один оператор, который фактически что-то делает, — оператор присваивания. Все другие операторы, такие как условные операторы и вызовы процедур, существуют только для того, чтобы управлять последовательностью выполнения операторов присваивания. К сожалению, трудно формально определить смысл оператора присваивания (в отличие от описания того, что происходит при его выполнении); фактически, вы никогда не встречали ничего подобного

при изучении математики в средней школе и колледже. То, что вы изучали, — были *уравнения:*

$$ax^2 + bx + c = 0$$

Вы преобразовывали уравнения, вы решали их и выполняли соответствующие вычисления. Но вы никогда их не изменяли: если *x* представляло число в одной части уравнения, то оно представляло то же самое число и в другой части уравнения.

Скромный оператор присваивания на самом деле является очень сложным и решает три разные задачи:

1. Вычисление значения выражения в правой части оператора.

2. Вычисление выражения в левой части оператора; выражение должно определять адрес ячейки памяти.

3. Копирование значения, вычисленного на шаге 1, в ячейки памяти, начиная с адреса, полученного на шаге 2.

Таким образом, оператор присваивания

```
a(i + 1) = b + c;
```

несмотря на внешнее сходство с уравнением, определяет сложное вычисление.

2.5. Контроль соответствия типов

В трехшаговом описании присваивания в результате вычисления выражения получается значение конкретного типа, в то время как вычисление левой части дает только начальный адрес блока памяти. Нет никакой гарантии, что адрес соответствует переменной того же самого типа, что и выражение; фактически, нет даже гарантии, что *размеры* копируемого значения и переменной совпадают.

Контроль соответствия типов — это проверка того, что тип выражения совместим с типом адресуемой переменной при присваивании. Сюда входит и присваивание фактического параметра формальному при вызове процедуры.

Возможны следующие подходы к контролю соответствия типов:

• Не делать ничего; именно программист отвечает за то, чтобы присваивание имело смысл.

• Неявно преобразовать значение выражения к типу, который требуется в левой части.

• *Строгий* контроль соответствия типов: отказ от выполнения присваивания, если типы различаются.

Существует очевидный компромисс между гибкостью и надежностью: чем строже контроль соответствия типов, тем надежнее будет программа, но потребуется больше усилий при программировании для определения подходящего набора типов. Кроме того, должна быть обеспечена возможность при необходимости обойти такой контроль. Наоборот, при слабом контроле соответствия типов проще писать программу, но зато труднее находить ошибки и гарантировать надежность программы. Недостаток контроля соответствия типов состоит в том, что его реализация может потребовать дополнительных затрат во время

выполнения программы. Неявное преобразование типов может оказаться хуже полного отсутствия контроля, поскольку при этом возникает ложная уверенность, что все в порядке.

Строгий контроль соответствия типов может исключить скрытые ошибки, которые обычно вызываются опечатками или недоразумениями. Это особенно важно в больших программных проектах, разрабатываемых группами программистов; из-за трудностей общения, смены персонала, и т.п. очень сложно объединять такое программное обеспечение без постоянной проверки, которой является строгий контроль соответствия типов. Фактически, строгий контроль соответствия типов пытается превратить ошибки, возникающие во время выполнения программы, в ошибки, выявляемые при компиляции. Ошибки, проявляющиеся только во время выполнения, часто чрезвычайно трудно найти, они опасны для пользователей и дорого обходятся разработчику программного обеспечения в смысле отсрочки сдачи программы и испорченной репутации. Цена ошибки компиляции незначительна: вам, вероятно, даже не требуется сообщать своему начальнику, что во время компиляции произошла ошибка.

2.6. Управляющие операторы

Операторы присваивания обычно выполняются в той последовательности, в какой они записаны. Управляющие операторы используются для изменения порядка выполнения. Программы на языке ассемблера допускают произвольные переходы по любым адресам. Язык программирования по аналогии может включать оператор goto, который осуществляет переход по метке на произвольный оператор. Программы, использующие произвольные переходы, трудно читать, а следовательно, изменять и поддерживать.

Структурное программирование — это название, данное стилю программирования, который допускает использование только тех управляющих операторов, которые обеспечивают хорошо структурированные программы, легкие для чтения и понимания. Есть два класса хорошо структурированных управляющих операторов.

• Операторы выбора, которые выбирают одну из двух или нескольких альтернативных последовательностей выполнения: условные операторы (if) и переключатели (case или switch).

• Операторы цикла, в которых повторяется выполнение последовательности операторов: операторы for и while.

Хорошее понимание циклов особенно важно по двум причинам: 1) большая часть времени при выполнении будет (очевидно) потрачена на циклы, и 2) многие ошибки связаны с неправильным кодированием начала или конца цикла.

2.7. Подпрограммы

Подпрограмма — это программный сегмент, состоящий из объявлений данных и исполняемых операторов, которые можно неоднократно *вызывать* (call) из различных частей программы. Подпрограммы называются *процедурами* (procedure), *функциями* (function), *подпрограммами* (subroutine) или *методами* (method). Первоначально они использовались только для того, чтобы разрешить многократное использование сегмента программы. Современная точка зрения состоит в том, что подпрограммы являются важным элементомструктуры программы и что каждый сегмент программы, который реализует некоторую конкретную задачу, следует оформить как отдельную подпрограмму.

При вызове подпрограммы ей передается последовательность значений, называемых *параметрами*. Параметры используются, чтобы задать различные варианты выполнения подпрограммы, передать ей данные и получить результаты вычисления. Механизмы передачи параметров сильно различаются в разных языках, и программисты должны хорошо понимать их воздействие на надежность и эффективность программы.

2.8. Модули

Тех элементов языка, о которых до сих пор шла речь, достаточно для написания *программ,* но недостаточно для написания *программной системы:* очень большой программы или набора программ, разрабатываемых группами программистов. Студенты часто на основе своих успехов в написании (небольших) программ заключают, что точно так же можно писать программные системы, но горький опыт показал, что написание большой системы требует дополнительных методов и инструментальных средств, выходящих за рамки простого программирования. Термин *проектирование программного обеспечения (software engineering)* используется для обозначения методов и инструментальных средств, предназначенных для проектирования, конструирования и управления при создании программных систем. В этой книге мы ограничимся обсуждением поддержки больших систем, которую можно осуществить с помощью языков программирования.

Возможно, вам говорили, что отдельная подпрограмма не должна превышать 40 или 50 строк, потому что программисту трудно читать и понимать большие сегменты кода. Согласно тому же критерию, должно быть понятно взаимодействие 40 или 50 подпрограмм. Отсюда следует очевидный вывод: любую программу, в которой больше 1600 — 2500 строк, трудно понять! Так как в полезных программах могут быть десятки тысяч строк и нередки системы из сотен тысяч строк, то очевидно, что необходимы дополнительные структуры для создания больших систем.

При использовании старых языков программирования единственным выходом были «бюрократические» методы: наборы правил и соглашений, которые предписывают членам группы, как следует писать программы. Современные языки программирования предлагают еще один метод структурирования для *инкапсуляции* данных и подпрограмм в более крупные объекты, называемые *модулями.* Преимущество модулей над бюрократическими предписаниями в том, что согласование модулей можно проверить при компиляции, чтобы предотвратить ошибки и недоразумения. Кроме того, фактически выполнимые операторы и большинство данных модуля (или все) можно скрыть таким образом, чтобы их нельзя было изменять или использовать, за исключением тех случаев, которые определены интерфейсом.

Есть две потенциальные трудности применения модулей на практике.

• Необходима мощная среда, разработки программ, чтобы отслеживать «истории», модулей и проверять интерфейсы. ;

• Разбиение на модули поощряет использование большого числа небольших подпрограмм с соответствующим увеличением времени выполнения из-за накладных расходов на вызовы подпрограмм.

Однако это больше не является проблемой: ресурсов среднего персонального компьютера более чем достаточно для поддержки среды языков C++ или Ada, а современная архитектура вычислительной системы и методы компиляции минимизируют издержки обращений.

Тот факт, что язык поддерживает модули, не помогает нам решать, что именно включить в модуль. Другими словами, остается вопрос, как разбить программную систему на модули? Поскольку качество системы непосредственно зависит от качества декомпозиции, компетентность разработчика программ должна оцениваться по способности анализировать требования проекта и создавать самую лучшую программную структуру для его реализации.

Требуется большой опыт, чтобы развить эту способность. Возможно, самый лучший способ состоит в том, чтобы изучать существующие системы программного обеспечения.

Несмотря на тот факт, что невозможно научить здравому смыслу в проектировании программ, есть некоторые принципы, которые можно изучить. Одним из основных методов декомпозиции программы является *объектно-ориентированное программирование* (ООП), опирающееся на концепцию типа, рассмотренную выше. Согласно ООП, модуль следует создавать для любого реального или абстрактного «объекта», который может представляться набором данных и операций над этими данными. В главах 14 и 15 детально обсуждается языковая поддержка ООП.

2.9. Упражнения

1. Переведите синтаксис (отдельные фрагменты) языков C или Ada из нормальной формы Бекуса — Наура в синтаксические диаграммы.

2. Напишите программу на языке Pascal или C, которая компилируется и выполняется, но вычисляет неправильный результат из-за незакрытого комментария.

3. Даже если бы язык Ada использовал стиль комментариев, как в языках C и Pascal, то ошибки, вызванные незакрытыми комментариями, были бы менее частыми, Почему?

4. В большинстве языков ключевые слова, подобные begin и while, зарезервированы и не могут использоваться как идентификаторы. В других языках типа Fortran и PL/1 нет зарезервированных ключевых слов. Каковы преимущества и недостатки зарезервированных слов?

Глава 3
Среды программирования

Язык — это набор правил для написания программ, которые являются всего лишь последовательностями символов. В этой главе делается обзор компонентов *среды программирования* — набора инструментов, используемых для преобразования символов в выполнимые вычисления.

Редактор - это, инструментальное средство для создания и изменения *исходных* файлов, которые являются символьными файлами, содержащими написанную на языке программирования программу.

Компилятор транслирует символы из исходного файла в *объектной модуль*, который содержит команды в машинном коде для конкретного комью-тера.

Библиотекарь поддерживает совокупности объектных файлов, называемые библиотеками.

Компоновщик, или редактор связей, собирает объектные файлы отдельных компонентов программы и *разрешает* внешние ссылки от одного компонента к другому, формируя исполняемый файл.

Загрузчик копирует исполняемый файл с диска в память и инициализирует компьютер перед выполнением программы.

Отладчик — это инструментальное средство, которое дает возможность программисту управлять выполнением программы на уровне отдельных операторов для диагностики ошибок.

Профилировщик измеряет, сколько времени затрачивается на каждый компонент программы. Программист, может затем улучшить эффективность *критических компонентов*, ответственных за большую часть времени выполнения.

Средства тестирования автоматизируют процесс тестирования программ, создавая и выполняя тесты и анализируя результаты тестирования.

Средства конфигурирования автоматизируют создание программ и прослеживают изменения до уровня исходных файлов.

Интерпретатор непосредственно выполняет исходный код программы в отличие от компилятора, переводящего исходный файл в объектный.

Среду программирования можно составить из отдельных инструментальных средств; кроме того, многие поставщики продают *интегрированные* среды программирования, которые представляют собой системы, содержащие большую часть или все перечисленные выше инструментальные средства. Преимущество интегрированной среды заключается в чрезвычайной простоте интерфейса пользователя: каждый инструмент инициируется нажатием единственной клавиши или выбором из меню вместо набора на клавиатуре имен файлов и параметров.

3.1. Редактор

У каждого программиста есть свой любимый универсальный редактор. Но даже в этом случае у вас может появиться желание воспользоваться специализированным, предназначенным для определенного Языка редактором, который создает всю языковую конструкцию, типа условного оператора, по одному нажатию клавиши. Преимущество такого редактора в том, что он позволяет предотвратить синтаксические ошибки. Однако любая машинистка, печатающая вслепую, скажет, что легче набирать языковые конструкции, чем отыскивать нужные пункты в меню.

В руководстве по языку могут быть указаны рекомендации для формата исходного кода: введение отступов, разбивка строк, использование верхнего/нижнего регистров. Эти правила не влияют на правильность программы, но ради будущих читателей вашей программы такие соглашения следует .соблюдать. Если вам не удалось выполнить соглашения при написании программы, то вы можете воспользоваться инструментальным средством, называемым *красивая печать* (pretty-printer), которое переформатирует исходный код к рекомендуемому формату. Поскольку эта программа может непреднамеренно внести ошибки, лучше соблюдать соглашения с самого начала.

3.2. Компилятор

Язык программирования без компилятора (или интерпретатора) может представлять большой теоретический интерес, но выполнить на компьютере программу, написанную на этом языке, невозможно. Связь между языками и компиляторами настолько тесная, что различие между ними расплывается, и часто можно услышать такую бессмыслицу, как:

Язык L1 эффективнее языка L2.

Правильно же то, что компилятор C1 может сгенерировать более эффективный код, чем компилятор C2, или что легче эффективно откомпилировать конструкции L1, чем соответствующие конструкции L2. Одна из целей этой книги — показать соотношение между конструкциями языка и получающимся после компиляции машинным кодом.

Структура компилятора показана на рис. *3.1. Входная часть* компилятора

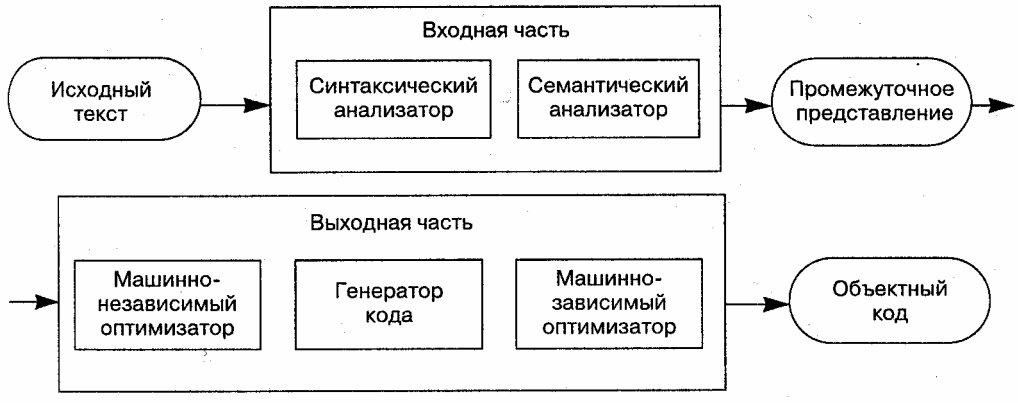

Рис. 3.1. Структура компилятора.

«понимает» программу, анализируя синтаксис и семантику согласно правилам языка. Синтаксический анализатор отвечает за преобразование последовательности символов в абстрактные синтаксические объекты, называемые *лексемами.* Например, символ «=» в языке С преобразуется в оператор присваивания, если за ним не следует другой «=»; в противном случае оба соседних символа «=» (т.е. «==») преобразуются в операцию проверки равенства. Анализатор семантики отвечает за придание смысла этим абстрактным объектам.

Например, в следующей программе семантический анализатор выделит глобальный адрес для первого i и вычислит смещение параметра — для второго i:

```
static int i;
void proc(inti) {... }
```

Результат работы входной части компилятора — абстрактное представление программы, которое называется *промежуточным представлением*. По нему можно восстановить исходный текст программы, за исключением имен идентификаторов и физического формата строк, пробелов, комментариев и т.д.

Исследования в области компиляторов настолько продвинуты, что входная часть может быть автоматически сгенерирована по *грамматике* (формальному описанию) языка. Читателям, интересующимся разработкой языков программирования, настоятельно рекомендуется глубоко изучить компиляцию и разрабатывать языки так, чтобы их было легко компилировать с помощью автоматизированных методов.

Выходная часть компилятора берет промежуточное представление программы и генерирует *машинный код* для конкретного компьютера. Таким образом, входная часть является *языковозависимой,* в то время как выходная — *машиннозависимой.* Поставщик компиляторов может получить семейство компиляторов некоторого языка L для ряда самых разных компьютеров Cl, C2,..., написав несколько выходных частей, использующих промежуточное представление общей входной части. Точно так же поставщик компьютеров может создать высококачественную выходную часть для компьютера С и затем поддерживать большое число языков LI, L2,..., написав входные части, которые компилируют исходный текст каждого языка в общее промежуточное представление. В этом случае фактически не имеет смысла спрашивать, какой язык на компьютере эффективнее.

С генератором объектного кода связан *оптимизатор,* который пытается улучшать код, чтобы сделать его более эффективным. Возможны несколько способов оптимизации:

• Оптимизация промежуточного представления, например нахождение общего подвыражения:

```
a = f1 (x + y) + f2(x + y);
```

Вместо того чтобы вычислять выражение x + y дважды, его можно вычислить один раз и сохранить во временной переменной или регистре. Подобная оптимизация не зависит от конкретного компьютера и может быть сделана до генерации кода. Это означает, что даже компоненты выходной части могут быть общими в компиляторах разных компьютеров.

• Машинно-ориентированная оптимизация. Такая оптимизация, как сохранение промежуточных результатов в регистрах, а не в памяти, явно должна выполняться при генерации объектного кода, потому что число и тип регистров в разных компьютерах различны.

• *Локальная оптимизация* обычно выполняется для сгенерированных команд, хотя иногда ее можно проводить для промежуточного представления. В этой методике делается попытка заменять короткие последовательности команд одной, более эффективной командой. Например, в языке С выражение n++ может быть скомпилировано в следующую последовательность:

```
load R1,n
add R1,#1
store R1,n
```

но локальный оптимизатор для конкретного компьютера мог бы заменить эти три команды одной, которая увеличивает на единицу непосредственно слово в памяти:

```
incr n
```

Использование оптимизаторов требует осторожности. Поскольку оптимизатор по определению изменяет программу, ее, возможно, будет трудно отлаживать с помощью отладчика исходного кода, так как порядок выполнения команд может отличаться от их порядка в исходном коде. Обычно оптимизатор при отладке лучше отключать. Кроме того, из-за сложности оптимизатора вероятность содержания в нем ошибки больше, чем в любом другом компоненте компилятора. Ошибку оптимизатора трудно обнаружить, потому что отладчик создан для работы с исходным текстом, а не с оптимизированным (то есть измененным) объектным кодом. Ни в коем случае нельзя сначала тестировать программу без оптимизатора, а после оптимизации отдавать в работу без тестирования. Наконец, оптимизатор в какой-либо ситуации может сделать неправильные предположения. Например, для устройства ввода-вывода с регистрами, «отображенными» на память, значение переменной может присваиваться дважды без промежуточного чтения:

```
transmit_register = 0x70; /* Ждать 1 секунду */ transmit_register = 0x70;
```

C

Оптимизатор предположит, что второе присваивание лишнее и удалит его из сгенерированного объектного кода.

3.3. Библиотекарь

Можно хранить объектные модули либо в отдельных файлах, либо в одном файле, называемом библиотекой. Библиотеки могут поставляться с компилятором, либо приобретаться отдельно, либо составляться программистом.

Многие конструкции языка программирования реализуются не с помощью откомпилированного кода, выполняемого внутри программы, а через обращения к процедурам, которые хранятся в библиотеке, предусмотренной поставщиком компилятора. Из-за увеличения объема языков программирования наблюдается тенденция к размещению большего числа функциональных возможностей в «стандартных» библиотеках, которые являются неотъемлемой частью языка. Так как библиотека — это всего лишь структурированная совокупность типов и подпрограмм, не содержащая новых языковых конструкций, то она упрощает задачи как для студента, который должен изучить язык, так и для разработчика компилятора.

Основной набор процедур, необходимых для инициализации, управления памятью, вычисления выражений и т.п., называется *системой времени исполнения (run-time system) или исполняющей системой.* Важно, чтобы программист был знаком с исполняющей системой используемого компилятора: невинные на первый взгляд конструкции языка могут фактически приводить к вызовам времяемких процедур в исполняющей системе. Например, если высокоточная арифметика реализована библиотечными процедурами, то замена всех целых чисел на длинные (двойные) целые значительно увеличит время выполнения.

3.4. Компоновщик

Вполне возможно написать программу длиной в несколько тысяч строк в виде отдельного файла или модуля. Однако для больших программных систем, особенно разрабатываемых группами программистов, требуется, чтобы программное обеспечение было разложено на модули (гл. 13). Если обращение делается к процедуре, находящейся вне текущего модуля, компилятор никак не может узнать адрес этой процедуры. Вместо этого адреса в объектном модуле записывается *внешняя ссылка*. Если язык разрешает разным модулям обращаться к глобальным переменным, то внешние ссылки должны быть созданы для каждого такого обращения. Когда все модули откомпилированы, компоновщик разрешает эти ссылки, разыскивая описания процедур и переменных, которые *экспортированы* из модуля для нелокального использования.

В современной практике программирования модули активно используются для декомпозиции программы. Дополнительный эффект такой практики — то, что компиляции бывают обычно короткими и быстрыми, зато компоновщик должен связать сотни модулей с тысячами внешних ссылок. Эффективность компоновщика может оказаться критичной для производительности группы разработчиков программного обеспечения на заключительных стадиях разработки: даже незначительное изменение одного исходного модуля потребует продолжительной компоновки.

Одно из решений этой проблемы состоит в том, чтобы компоновать из модулей подсистемы и только затем разрешать связи между подсистемами. Другое решение состоит в использовании *динамической компоновки*, если она поддерживается системой. При динамической компоновке внешние ссылки не разрешаются; вместо этого операционной системой улавливается и разрешается первое обращение к процедуре. Динамическая компоновка может комбинироваться с *динамической загрузкой*, при этом не только ссылки не разрешаются, но даже модуль не загружается, пока не понадобится одна из экспортируемых им процедур. Конечно, динамическая компоновка или загрузка приводит к дополнительным издержкам во время выполнения, но это мощный метод адаптации систем к изменяющимся требованиям без перекомпоновки.

3.5. Загрузчик

Как подразумевает название, загрузчик загружает программу в память и инициализирует ее выполнение. На старых компьютерах загрузчик был не-тривиален, так как должен был решать проблему *перемещаемости* программ. Такая команда, как load 140, содержала абсолютный адрес памяти, и его приходилось настраивать в зависимости от конкретных адресов, в которые загружалась программа. В современных компьютерах адреса команд и данных задаются относительно значений в регистрах. Для каждой области памяти с программой или данными выделяется регистр, указывающий на начало этой области, поэтому все, что должен сделать загрузчик теперь, — это скопировать программу в память и инициализировать несколько регистров. Команда load 140 теперь означает «загрузить значение, находящееся по адресу, полученному прибавлением 140 к содержимому регистра, который указывает на область данных».

3.6. Отладчик

Отладчики поддерживают три функции.

Трассировка. Пошаговое выполнение программы, позволяющее программисту точно отслеживать команды в порядке их выполнения.

Контрольные точки. Средство, предназначенное для того, чтобы заставить программу выполняться до конкретной строки в программе. Специальный вид контрольной точки — *точка наблюдения* — вызывает выполнение программы, пока не произойдет обращение к определенной ячейке памяти.

Проверка/изменение данных. Возможность посмотреть и изменить значение любой переменной в любой точке вычисления.

Символьные отладчики работают с символами исходного кода (именами переменных и процедур), а не с абсолютными машинными адресами. Символьный отладчик требует взаимодействия компилятора и компоновщика для того, чтобы создать таблицы, связывающие символы и их адреса.

Современные отладчики чрезвычайно мощны и гибки. Однако ими не следует злоупотреблять там, где надо подумать. Часто несколько дней трассировки дают для поиска ошибки меньше, чем простая попытка объяснить процедуру другому программисту.

Некоторые проблемы трудно решить даже с помощью отладчика. Например, динамические структуры данных (списки и деревья) нельзя исследовать в целом; вместо этого нужно вручную проходить по каждой связи. Есть более серьезные проблемы типа затирания памяти (см. раздел 5.3), которые вызваны ошибками, находящимися далеко от того места, где они проявились. В этих ситуациях мало проку от отладчиков, нацеленных на выявление таких симптомов, как «деление на ноль в процедуре p1».

Наконец, некоторые системы не могут быть «отлажены» как таковые: нельзя по желанию создать тяжелое положение больного только для того, чтобы отладить программное обеспечение сердечного монитора; нельзя послать группу программистов в космический полет для того, чтобы отладить управляющую программу полета. Такие системы должны проверяться с помощью специальных аппаратных средств и программного обеспечения для моделирования входных и выходных данных; программное обеспечение в таких случаях никогда не проверяется и не отлаживается в реальных условиях! Программные системы, критичные в отношении надежности, стимулируют исследование языковых конструкций, повышающих надежность программ и вносящих вклад в формальные методы их верификации.

3.7. Профилировщик

Часто говорят, что попытки улучшить эффективность программы вызывают больше ошибок, чем все другие причины. Этот вывод столь пессимистичен из-за того, что большинство попыток улучшения эффективности ни к чему хорошему не приводят или в лучшем случае позволяют добиться усовершенствований, которые несоразмерны затраченным усилиям. В этой книге мы обсудим относительную эффективность различных программных конструкций, но этой информацией стоит воспользоваться только при выполнении трех условий:

• Текущая эффективность программы неприемлема.

• Не существует лучшего способа улучшить эффективность. В общем случае выбор более эффективного алгоритма даст лучший результат, чем попытка перепрограммировать существующий алгоритм (для примера см. раздел 6.5).

• Можно выявить причину неэффективности.

Чрезвычайно сложно обнаружить причину неэффективности без помощи измерительных средств. Дело в том, что временные интервалы, которые мы инстинктивно воспринимаем (секунды), и временные интервалы работы компьютера (микро- или наносекунды) отличаются на порядки. Функция, которая нам кажется сложной, возможно, оказывает несущественное влияние на общее время выполнения программы.

Профилировщик периодически опрашивает указатель выполняемой команды компьютера и затем строит гистограмму, отображающую процент времени выполнения для каждой процедуры или команды. Очень часто результат удивляет программиста, выявляя узкие места, которые совсем не были очевидны. Крайне непрофессионально выполнять оптимизацию программы без использования профилировщика.

Даже с профилировщиком может оказаться трудно улучшить эффективность программы. Одна из причин состоит в том, что большая часть времени выполнения тратится в получаемых извне компонентах программы, таких как базы данных или подсистемы работы с окнами, которые часто разрабатываются больше по критериям гибкости, чем эффективности.

3.8. Средства тестирования

Тестирование большой системы может занять столько же времени, сколько и программирование вместе с отладкой. Для автоматизации отдельных аспектов тестирования были разработаны программные инструментальные средства. Одно из них — *анализатор покрытия (coverage analyzer)*, который отслеживает, какие команды были протестированы. Однако такой инструмент не помогает создавать и выполнять тесты.

Более сложные инструментальные средства выполняют заданные Тесты, а затем сравнивают вывод со спецификацией. Тесты также могут генерироваться автоматически, фиксируя ввод с внешнего источника вроде нажатия пользователем клавиш на клавиатуре. Зафиксированную входную последовательность затем можно выполнять неоднократно. Всякий раз при выпуске новой версии программной системы следует предварительно снова запускать тесты. Такое *регрессивное тестирование* необходимо, потому что предположения, лежащие в основе программы, настолько взаимосвязаны, что любое изменение может вызвать ошибки даже в модулях, где «ничего не изменялось».

3.9. Средства конфигурирования

Инструментальные средства конфигурирования используются для автоматизации управленческих задач, связанных с программным обеспечением. Инструмент *сборки (make)* создает исполняемый файл из исходных текстов, вызывая компилятор, компоновщик, и т.д. При проектировании большой системы может оказаться трудно в точности отследить, какие файлы должны быть перекомпилированы, в каком порядке и с какими параметрами, и легко, найдя и исправив ошибку, тут же вызвать другую, использовав устаревший объектный модуль. Инструмент сборки программы гарантирует, что новый исполняемый файл создан корректно с минимальным количеством перекомпиляций.

Инструментальные средства *управления исходными текстами (source control)* или *управления изменениями (revision control)* используются для отслеживания и регистрации всех изменений модулей исходного текста. Это важно, потому что при проектировании больших систем часто необходимо отменить изменение, которое вызвало непредвиденные проблемы, либо проследить изменения для конкретной версии или сделанные конкретным

программистом. Кроме того, разным заказчикам могут поставляться различные версии программы, а без программных средств пришлось бы устранять общую ошибку во всех версиях. Инструментальные средства управления изменениями упрощают эти задачи, поскольку сохраняют изменения (так называемые *дельты)* относительно первоначальной версии и позволяют на их основе легко восстановить любую предыдущую версию.

3.10. Интерпретаторы

Интерпретатор — это программа, которая непосредственно выполняет код исходной программы. Преимущество интерпретатора перед компилятором состоит в чрезвычайной простоте использования, поскольку не нужно вызывать всю последовательность инструментальных средств: компилятор, компоновщик, загрузчик, и т.д. К тому же интерпретаторы легко писать, поскольку они могут не быть машинно-ориентированными; они непосредственно выполняют программу, и у них на выходе нет никакого машинного кода. Таким образом, интерпретатор, написанный на стандартизированном языке, является переносимым. Относительная простота интерпретаторов связана также с тем, что они традиционно не пытаются что-либо оптимизировать.

В действительности провести различие между интерпретатором и компилятором бывает трудно. Очень немногие интерпретаторы действительно выполняют исходный код программы; вместо этого они переводят (то есть компилируют) исходный код программы в код некой воображаемой машины и затем выполняют абстрактный код (рис. 3.2).

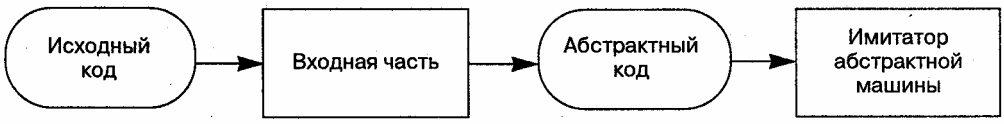

Рис. 3.2. Структура интерпретатора.

Предположим теперь, что некто изобрел компьютер с машинным кодом, в точности совпадающим с этим абстрактным кодом; или же предположим, что некто написал набор макрокоманд, которые заменяют абстрактный машинный код фактическим машинным кодом реального компьютера. В любом случае так называемый интерпретатор превращается в компилятор, не изменившись при этом ни в одной своей строке.

Первоначально Pascal-компилятор был написан для получения машинного кода конкретной машины (CDC 6400). Немного позже Никлаус Вирт создал компилятор, который вырабатывал код, названный P-кодом, для абстрактной стековой машины. Написав интерпретатор для P-кода или компилируя P-код в машинный код конкретной машины, можно создать интерпретатор или компилятор для языка Pascal, затратив относительно небольшие усилия. Компилятор для P-кода был решающим фактором в превращении языка Pascal в широко распространенный язык, каким он является сегодня.

Язык логического программирования Prolog (см. гл. J7) рассматривался вначале как язык, пригодный только для интерпретации. Дэвид Уоррен (David Warren) создал первый настоящий компилятор для языка Prolog, описав абстрактную машину (абстрактная машина Уоррена, или WAM), которая управляла основными структурами данных, необходимыми для выполнения программы на языке. Как компиляцию Prolog в WAM-программы, так и компиляцию WAM-программы в машинный код проделать не слишком трудно; достижение Уоррена состояло в том, что он сумел между двух уровней определить правильный промежуточный уровень — уровень WAM. Многие исследования по компиляции языков логического программирования опирались на WAM.

Похоже, что споры о различиях между компиляторами и интерпретаторами не имеют большого практического значения. При сравнении сред программирования особое внимание следует уделять надежности, правильной компиляции, высокой производительности,

эффективной генерации объектного кода, хорошим средствам отладки и т.д., а не технологическим средствам создания самой среды.

3.11. Упражнения

1. Изучите документацию используемого вами компилятора и перечислите оптимизации, которые он выполняет. Напишите программы и проверьте получающийся в результате объектный код на предмет оптимизации.

2. В какой информации от компилятора и от компоновщика нуждается отладчик?

3. Запустите профилировщик и изучите, как он работает.

4. Как можно написать собственный простой инструментарий для тестирования? В чем заключается влияние автоматизированного тестирования на проектирование программы?

5. AdaS — написанный на языке Pascal интерпретатор для подмножества Ada. Он работает, компилируя исходный код в P-код и затем выполняя P-код. Изучите AdaS-программу (см. приложение A) и опишите P-ма-шину.

2 Основные понятия

Глава 4
Элементарные типы данных

4.1. Целочисленные типы

Слово *«целое» (integer) в* математике обозначает неограниченную, упорядоченную последовательность чисел:

...,-3, -2,-1,0,1,2,3,...

В программировании этот термин используется для обозначения совсем другого — особого типа данных. Вспомним, что тип данных — это множество значений и набор операций над этими значениями. Давайте начнем с определения множества значений типа Integer (целое).

Для слова памяти мы можем определить множество значений, просто интерпретируя биты слова как двоичные значения. Например, если слово из 8 битов содержит последовательность 10100011, то она интерпретируется как:
$(1 \times 2^7) + (1 \times 2^5) + (1 \times 2^1) + (1 \times 2°) = 128 + 32 + 2 + 1 = 163$

Диапазон возможных значений — 0.. 255 или в общем случае 0.. 2^B - 1 для слова из B битов. Тип данных с этим набором значений называется *unsigned integer (целое без знака),* а переменная этого типа может быть объявлена в языке C как:

unsigned intv;

Обратите внимание, что число битов в значении этого типа может быть разным для разных компьютеров.

Сегодня чаще всего встречается размер слова в 32 бита, и целое (без знака) находится в диапазоне 0.. 2^{32} - 1 к 4 x 10^9. Таким образом, набор математических целых чисел неограничен, в то время как целочисленные типы имеют конечный диапазон значений.

Поскольку тип unsigned integer не может представлять отрицательные числа, он часто используется для представления значений, считываемых внешними устройствами.

Например, при опросе температурного датчика поступает 10 битов информации; эти целые без знака в диапазоне 0.. 1023 нужно будет затем преобразовать в обычные (положительные и отрицательные) числа. Целые числа без знака также используются для представления символов (см. ниже). Их не следует использовать для обычных вычислений, потому что большинство компьютерных команд работает с целыми числами со знаком, и компилятор, возможно, будет генерировать дополнительные команды для операций с целыми без знака.

Диапазон значений переменной может быть таким, что значения не поместятся в одном слове или займут только часть слова. Чтобы указать разные целочисленные типы, можно добавить спецификаторы длины:

```
unsigned int v1 ;              /* обычное целое */          [C]
unsigned short int v2;         /* короткое целое */
unsigned long int v3;          /* длинное целое */
```

В языке Ada наряду с обычным типом Integer встроены дополнительные типы, например Long_integer (длинное целое). Фактическая интерпретация спецификаторов длины, таких как long и short, различается для различных компиляторов; некоторые компиляторы могут даже давать одинаковую интерпретацию двум или нескольким спецификаторам.

В математике для представления чисел со знаком используется специальный символ «-», за которым следует обычная запись абсолютного значения числа. Компьютеру с таким представлением работать неудобно. Поэтому большинство компьютеров представляет целые числа со знаком в записи, называющейся *дополнением до двух* *. Положительное число представляется старшим нулевым битом и следующим за ним обычным двоичным представлением значения. Из этого вытекает, что самое большое положительное целое число, которое может быть представлено словом из w битов, не превышает 2^{W-1} - 1.

Для того чтобы получить представление числа -n по двоичному представлению B = b1b2...b_wчисла n:

• берут логическое *дополнение* B, т. е. заменяют в каждом b ноль на единицу, а единицу на ноль,

• прибавляют единицу.

Например, представление -1, -2 и -127 в виде 8-разрядных слов получается так:

```
  1  =  0000 0001 →  1111 1110  →  1111 1111 =   -1
  2  =  0000 0010 →  1111 1101  →  1111 1110 =   -2
127  =  0111 1111 →  1000 0000  →  1000 0001 = -127
```

У отрицательных значений в старшем бите всегда будет единица.

Дополнение до двух удобно тем, что при выполнении над такими представлениями операций обычной двоичной целочисленной арифметики получается правильное представление результата:

(-!)-! = -2
1111 1111-00000001 = 1111 1110

Отметим, что строку битов 1000 0000 нельзя получить ни из какого положительного значения. Она представляет значение -128, тогда как соответствующее положительное значение 128 нельзя представить как 8-разрядное число. Необходимо учитывать эту асимметрию в диапазоне типов integer, особенно при работе с типами short.

Альтернативное представление чисел со знаками — *дополнение до единицы*, в котором представление значения -n является просто дополнением n. В этом случае набор значений симметричен, но зато есть два представления для нуля: 0000 0000 называется положительным нулем, а 1111 1111 называется отрицательным нулем.

Если в объявлении переменной синтаксически не указано, что она без знака (например, unsigned), то по умолчанию она считается целой со знаком:

I

```
nt i;              /* Целое со знаком в языке C */
I: Integer;        -- Целое со знаком в языке Ada
```

Целочисленные операции

К целочисленным операциям относятся четыре основных действия: сложение, вычитание, умножение и деление. Их можно использовать для составления *выражений:*

a + b/c - 25* (d - e)

К целочисленным операциям применимы обычные математические правила старшинства операций; для изменения порядка вычислений можно использовать круглые скобки.

Результат операции над целыми числами со знаком не должен выходить за диапазон допустимых значений, иначе произойдет переполнение, как рассмотрено ниже. Для целых чисел без знака используется циклическая арифметика. Если short int хранится в 16-разрядном слове, то:

C

```
unsigned short int i;          /* Диапазон i= 0...65535*/
i = 65535;                     /* Наибольшее допустимое значение*/
i = i + 1;                     /*Циклическая арифметика, i = 0 */
```

Разработчики Ada 83 сделали ошибку, не включив в язык целые без знака. Ada 95 обобщает концепцию целых чисел без знака до *модульных типов*, которые являются целочисленными типами с циклической арифметикой по произвольному модулю. Обычный байт без знака можно объявить как:

Ada

```
type Unsigned_Byte is mod 256;
```

тогда как модуль, не равный двум, можно использовать для хеш-таблиц или случайных чисел:

Ada

```
Ada type Randomjnteger is mod 41;
```

Обратите внимание, что модульные типы в языке Ada переносимы, так как частью определения является только циклический диапазон, а не *размер* представления, как в языке С.

Деление

В математике в результате деления двух целых чисел *a/b* получаются два значения: *частное q* и *остаток r,* такие что:

$a = q * b + r$

Так как результатом арифметического выражения в программах является единственное значение, то для получения частного используют оператор «/», а для получения остатка применяют другой оператор (в языке С это «%», а в Ada — rem). Выражение 54/10 дает значение 5, и мы говорим, что результат операции был *усечен (truncated).* В языке Pascal для целочисленного деления используется специальная операция div.

При рассмотрении отрицательных чисел определение целочисленного деления не столь тривиально. Чему равно выражение -54/10: -5 или -6? Другими словами, до какого значения делается усечение: до меньшего («более отрицательного») или до ближайшего к нулю? Один вариант — это усечение в сторону нуля, поскольку, чтобы удовлетворить соотношение для целочисленного деления, достаточно просто сменить знак остатка:

-54 = -5*10 + (-4)

Однако существует и другая математическая операция, *взятие по модулю (modulo),* которая соответствует округлению отрицательных частных до меньшего («более отрицательного») значения:

$$-54 = -6 * 10 + 6$$
$$-54 \bmod 10 = 6$$

Арифметика по модулю используется всякий раз, когда арифметические операции выполняются над конечными диапазонами, например над кодами с исправлением ошибок в системах связи.

Значение операций «/» и «%» в языке C зависит от реализации, поэтому программы, использующие эти целочисленные операции, могут оказаться непереносимыми. В Ada операция «/» всегда усекает в сторону нуля. Операция rem возвращает остаток, соответствующий усечению в сторону нуля, в то время как операция mod возвращает остаток, соответствующий усечению в сторону минус бесконечности.

Переполнение

Говорят, что операция приводит к *переполнению,* если она дает результат, который выходит за диапазон допустимых значений. Следующие рассуждения для ясности даются в терминах 8-разрядных целых чисел.

Предположим, что переменная i типа signed integer имеет значение 127 и что мы увеличиваем i на 1. Компьютер просто прибавит единицу к целочисленному представлению 127:

$$0111\ 1111 + 00000001 = 10000000$$

и получит -128. Это неправильный результат, и ошибка вызвана переполнением. Переполнение может приводить к странным ошибкам:

C

```
for (i = 0; i < j*k; i++)….
```

Если происходит переполнение выражения j*k, верхняя граница может оказаться отрицательной и цикл не будет выполнен.

Предположите теперь, что переменные a, b и c имеют значения 90, 60 и 80, соответственно. Выражение (a - b + c) вычисляется как 110, потому что (a - b) дает 30, и затем при сложении получается 110. Однако оптимизатор для вычисления выражения может выбрать другой порядок, (a + c - b), давая неправильный ответ, потому что сложение (a + c) дает значение 170, что вызывает переполнение. Если вам в средней школе говорили, что сложение является коммутативным и ассоциативным, то речь шла о математике, а не о программировании!

В некоторых компиляторах можно задать режим проверки на переполнение каждой целочисленной операции, но это может привести к чрезмерным издержкам времени выполнения, если обнаружение переполнения не делается аппаратными средствами. Теперь, когда большинство компьютеров использует 32-разрядные слова памяти, целочисленное переполнение встречается редко, но вы должны знать о такой возможности и соблюдать осторожность, чтобы не попасть в ловушки, продемонстрированные выше.

Реализация

Целочисленные значения хранятся непосредственно в словах памяти. Некоторые компьютеры имеют команды для вычислений с частями слов или даже отдельными байтами. Компиляторы для этих компьютеров обычно помещают short int в часть слова, в то время как компиляторы для компьютеров, которые распознают только полные слова, реализуют целочисленные типы int и short int одинаково. Тип long int обычно распределяется в два слова, чтобы получить больший диапазон значений.

Сложение и вычитание компилируются непосредственно в соответствующие команды. Умножение также превращается в одну команду, но выполняется значительно дольше, чем сложение и вычитание. Умножение двух слов, хранящихся в регистрах R1 и R2, дает результат длиной в два слова и требует для хранения двух регистров. Если регистр, содержащий старшее значение, — не ноль, то произошло переполнение.

Для деления требуется, чтобы компьютер выполнил итерационный алгоритм, аналогичный «делению в столбик», выполняемому вручную. Это делается аппаратными средствами, и вам не нужно беспокоиться о деталях, но, если для вас важна эффективность, деления лучше избегать.

Арифметические операции выполняются для типа long int более чем вдвое дольше, нежели операции для int. Причина в том, что нужны дополнительные команды для «распространения» переноса, который может возникать, из слова младших разрядов в слово старших.

4.2. Типы перечисления

Языки программирования типа **Fortran** и **C** описывают данные в терминах компьютера. Данные реального мира должны быть явно отображены на типы данных, которые существуют на компьютере, в большинстве случаев на один из целочисленных типов. Например, если вы пишете программу для управления нагревателем, вы могли бы использовать переменную dial для хранения текущей позиции регулятора. Предположим, что реальная шкала имеет четыре позиции: *off (выключено), low (слабо), medium (средне), high (сильно).* Как бы вы объявили переменную и обозначили позиции? Поскольку компьютер не имеет команд, которые работают со словами памяти, имеющими только четыре значения, для объявления переменной вы выберете тип integer, а для обозначения позиций четыре конкретных целых числа (скажем 1, 2, 3, 4):

C

```
int dial;                    /* Текущая позиция шкалы */
 if (dial < 4) dial++;       /* Увеличить уровень нагрева*/
```

Очевидно, что при использовании целых чисел программу становится трудно читать и поддерживать. Чтобы понять код программы, вам придется написать обширную документацию и постоянно в нее заглядывать. Чтобыулучшить программу, можно прежде всего задокументировать подразумеваемые значения внутри самой программы:

C

```
 #define Off          1
#define Low           2
#define Medium        3
#define High          4

int dial;
if(dial<High)dial++;
```

Однако улучшение документации ничего не дает для предотвращения следующих проблем:

```
dial=-1;            /* Нет такого значения*/
dial = High + 1;    /* Нераспознаваемое переполнение*/
dial = dial * 3;    /* Бессмысленная операция*/
```

C

Другими словами, представление шкалы с четырьмя позициями в виде целого числа позволяет программисту присваивать значения, которые выходят за допустимый диапазон, и выполнять команды, бессмысленные для реального объекта. Даже если программист не создаст преднамеренно ни одну из этих проблем, опыт показывает, что они часто появляются в результате отсутствия I взаимопонимания между членами группы разработчиков программы, опеча- ток и других ошибок, типичных при создании сложных систем.

Решение состоит в том, чтобы разрешить разработчику программы созда- вать *новые* типы, точно соответствующие тем объектам реального мира, которые нужно моделировать. Рассматриваемая здесь короткая упорядоченная последовательность значений настолько часто встречается, что современные языки программирования поддерживают создание типов, называемых *типами — перечислениями (enumiration types)**. В языке Ada вышеупомянутый пример выглядел бы так:

```
Ada type Heat is (Off, Low, Medium, High);
Dial: Heat;
Dial := Low;
if Dial < High then Dial := Heat'Succ(DialJ;
Dial:=-1;                                  —Ошибка
Dial := Heat'Succ(High);                    -- Ошибка
Dial := Dial * 3;                           - Ошибка
```

Ada

Перед тем как подробно объяснить пример, обратим внимание на то, что в языке С есть конструкция, на первый взгляд точно такая же:

```
typedef enum {Off, Low, Medium, High} Heat;
```

C

Однако переменные, объявленные с типом Heat, — все еще целые, и ни одна из вышеупомянутых команд не считается ошибкой (хотя компилятор может выдавать предупреждение):

```
Heat dial;

dial = -1;              /*Не является ошибкой!*/
dial = High + 1;        /* Не является ошибкой! */
dial = dial * 3;        /* Не является ошибкой! */
```

C

Другими словами, конструкция enum* — всего лишь средство документирования, более удобное, чем длинные строки define, но она не создает новый тип.

К счастью, язык С++ использует более строгую интерпретацию типов перечисления и не допускает присваивания целочисленного значения переменной перечисляемого типа; указанные три команды здесь будут ошибкой. Однако значения перечисляемых типов могут быть неявно преобразованы в целые числа, поэтому контроль соответствия типов не является полным. К сожалению, в С++ не предусмотрены команды над перечисляемыми типами, поэтому здесь нет стандартного способа увеличения переменной этого типа. Вы можете написать свою собственную функцию, которая берет результат целочисленного выражения и затем явно преобразует его к типу перечисления:

```
dial = (Heat) (dial + 1);
```

C++

Обратите внимание на неявное преобразование dial в целочисленный тип, вслед за которым происходит явное преобразование результата обратно в Heat. Операции «++» и «--» над целочисленными типами в C++ можно перегрузить (см. раздел 10.2), поэтому они могут быть использованы для определения операций над типами перечисления, которые синтаксически совпадают с операциями над целочисленными типами.

В языке Ada определение типа приводит к созданию нового типа Heat. Значения этого типа *не являются целыми числами*. Любая попытка выйти за диапазон допустимых значений или применить целочисленные операции будет отмечена как ошибка. Если вы случайно нажмете не на ту клавишу и введете Higj вместо High, ошибка будет обнаружена, потому что тип содержит именно те четыре значения, которые были объявлены. Если бы вы использовали один из типов integer, 5 было бы допустимым целым, как и 4.

Перечисляемые типы аналогичны целочисленным: вы можете объявлять переменные и параметры этих типов. Однако набор операций, которые могутвыполняться над значениями этого типа, ограничен. В него входят присваивание (:=), равенство (=) и неравенство (/=). Поскольку набор значений в объявлении интерпретируется как упорядоченная последовательность, для него определены операции отношений (<,>,>=,<=).

В языке Ada для заданного T перечисляемого типа и значения V типа T определены следующие функции, называемые *атрибутами:*

• T'First возвращает первое значение T.

• T'Last возвращает последнее значение T.

• T'Succ(V) возвращает следующий элемент V.

• T'Pred(V) возвращает предыдущий элемент V.

• T'Pos(V) возвращает позицию V в списке значений T.

• T'Val(l) возвращает значение I-й позиции в T.

Атрибуты делают программу устойчивой к изменениям: при добавлении значений к типу перечисления или переупорядочивании значений циклы и индексы остаются неизменными:

```
for I in Heat'First.. Heat'Last - 1 loop
A(l):=A(Heat'Succ(l));
end loop;
```

Ada

Не каждый разработчик языка «верует» в перечисляемые типы. В языке Eiffel их нет по следующим причинам:

• Желательно было сделать язык как можно меньшего объема.

• Можно получить тот же уровень надежности, используя контрольные утверждения (раздел 11.5).

• Перечисляемые типы часто используются с вариантными записями (раздел 10.4); при правильном применении наследования (раздел 14.3) потребность в перечисляемых типах уменьшается.

Везде, где только можно, следует предпочитать типы перечисления обычным целым со списками заданных констант; их вклад в надежность программы невозможно переоценить. Программисты, работающие на C, не имеют преимуществ контроля соответствия типов, как в Ada и C++, и им все же следует использовать enum, чтобы улучшить читаемость программы.

Реализация

Я расскажу вам по секрету, что значения перечисляемого типа представляются в компьютере в виде последовательности целых чисел, начинающейся с нуля. Контроль соответствия типов в языке Ada делается только во время компиляции, а такие операции как «<» представляют собой обычные целочисленные операции.

Можно потребовать, чтобы компилятор использовал нестандартное представление перечисляемых типов. В языке C это задается непосредственно в определении типа:

typedef enum {Off = 1, Low = 2, Medium = 4, High = 8} Heat; C

тогда как в Ada используется спецификация представления: __

type Heat is (Off, Low, Medium, High); Ada
for Heat use (Off = >1, Low = >2, Medium = >4, High = >8);

4.3. Символьный тип

Хотя первоначально компьютеры были изобретены для выполнения операций над числами, скоро стало очевидно, что не менее важны прикладные программы для обработки нечисловой информации. Сегодня такие приложения, как текстовые процессоры, образовательные программы и базы данных, возможно, по количеству превосходят математические прикладные программы. Даже такие математические приложения, как финансовое программное обеспечение, нуждаются в обработке текста для ввода и вывода.

С точки зрения разработчика программного обеспечения обработка текста чрезвычайно сложна из-за разнообразия естественных языков и систем записи. С точки зрения языков программирования обработка текста относительно проста, так как подразумевается, что в языке набор символов представляет собой короткую, упорядоченную последовательность значений, то есть символы могут быть определены перечисляемым типом. Фактически, за исключением языков типа китайского и японского, в которых используются тысячи символов, достаточно 128 целых значений со знаком или 256 значений без знака, представимых восемью разрядами.

Различие в способе определения символов в языках Ada и C аналогично различию в способе определения перечисляемых типов. В Ada есть встроенный перечисляемый тип: __

type Character is (..., 'A', 'B',...); Ada

и все обычные операции над перечисляемыми типами (присваивание, отношения, следующий элемент, предыдущий элемент и т.д.) применимы к символам. В Ada 83 для типа Character допускались 128 значений, определенных в американском стандарте ASCII, в то время как в Ada 95 принято представление этого типа байтом без знака, так что доступно 256 значений, требуемых международными стандартами.

В языке C тип char — это всего лишь ограниченный целочисленный тип, и допустимы все следующие операторы, поскольку char и int по сути одно и то же:

```
char c;
int i;
c='A' + 10;          /* Преобразует char в int и обратно */
i = 'A';             /* Преобразует char в int */
c = i;               /* Преобразует int в char */
```

C

В языке C++ тип char отличается от целочисленного, но поскольку допустимы преобразования в целочисленный и обратно, то перечисленные операторы остаются допустимыми.

Для неалфавитных языков могут быть определены 16-разрядные символы. Они называются wcharj в C и C++, и Wide_Character в Ada 95.

Единственное, что отличает символы от обычных перечислений или целых, — специальный синтаксис ('A') для набора значений и, что более важно, специальный синтаксис для массивов символов, называемых *строками* (раздел 5.5).

4.4. Булев тип

Boolean — встроенный перечисляемый тип в языке Ada:

```
type Boolean is (False, True);
```

Тип Boolean имеет очень большое значение, потому что:

• операции отношения (=, >, и т.д.) — это функции, которые возвращают значение булева типа;

• условный оператор проверяет выражение булева типа;

• операции булевой алгебры (and, or, not, xor) определены для булева типа.

В языке C нет самостоятельного булева типа; вместо этого используются целые числа в следующей интерпретации:

• Операции отношения возвращают 1, если отношение выполняется, и 0 в противном случае.

• Условный оператор выполняет переход по ветке false (ложь), если вычисление целочисленного выражения дает ноль, и переход по ветке true (истина) в противном случае.

В языке C существует несколько методов введения булевых типов. Одна из возможностей состоит в определении типа, в котором будет разрешено объявление функций с результатом булева типа:

```
typedef enum {false, true} bool;
bool data_valid (int a, float b);
if (data-valid (x, y)). . .
```

C

но это применяется, конечно, только для документирования и удобочитаемости, потому что такие операторы, как:

```
bool b;                                                    C
b = b + 56;          /* Сложить 56 с «true» ?? */
```

все еще считаются приемлемыми и могут приводить к скрытым ошибкам.

В языке C++ тип bool является встроенным *целочисленным* типом (не типом перечисления) с неявными взаимными преобразованиями между ненулевыми значениями и литералом true, а также между нулевыми значениями и false. Программа на C с bool, определенным так, как показано выше, может быть скомпилирована на C++ простым удалением typedef.

Даже в языке C лучше не использовать неявное преобразование целых в булевы, а предпочитать явные операторы равенства и неравенства:

```
if (a + b== 2)...          /* Этот вариант понятнее, чем */          C
if (a + b-2)...            /* ...такойвариант.*/
if (a + b ! = O)...        /* Этот вариант понятнее, чем */
if (! (a + b))...          /*... такой вариант. */
```

Наконец, отметим, что в языке C применяется так называемое укороченное (short-circuit) вычисление выражений булевой алгебры. Это мы обсудим в разделе 6.2.

4.5. Подтипы

В предыдущих разделах мы обсудили целочисленные типы, позволяющие выполнять вычисления в большом диапазоне значений, которые можно представить в слове памяти, и перечисляемые типы, которые работают с меньшими диапазонами, но не позволяют выполнять арифметические вычисления. Однако во многих случаях нам хотелось бы делать вычисления в небольших диапазонах целых чисел. Например, должен существовать какой-нибудь способ, позволяющий обнаруживать такие ошибки как:

```
Temperature: Integer;
Temperature := -280;                    -- Ниже абсолютного нуля!
Compass-Heading: Integer;
Compass-Heading := 365;                  - Диапазон компаса 0..359 градусов!
```

Предположим, что мы попытаемся определить новый класс типов:

```
 type Temperatures is Integer range -273 .. 10000;   - - He Ada!
 type Headings is Integer range 0 .. 359;             -- He Ada!
```

Это решает проблему проверки ошибок, вызванных значениями, выходящими за диапазон типа, но остается вопрос: являются эти два типа разными или нет? Если это один и тот же тип, то

```
Temperature * Compass_Heading
```

является допустимым арифметическим выражением на типе *целое;* если нет, то должно использоваться преобразование типов.

На практике полезны обе эти интерпретации. В вычислениях, касающихся физического мира, удобно использовать значения разных диапазонов. С другой стороны, индексы в таблицах или порядковые номера не требуют вычислений с разными типами: имеет смысл запросить следующий индекс в таблице, а не складывать индекс таблицы с порядковым номером. Для этих двух подходов к определению типов в языке Ada есть два разных средства.

Подтип (subtype) — это ограничение на существующий тип. Дискретные типы (целочисленные и перечисляемые) могут иметь *ограничение диапазона*.

```
subtype Temperatures is Integer range -273 .. 10000;
Temperature: Temperatures;

subtype Headings is Integer range 0 .. 359;
Compass_Heading: Headings;
```

Тип значения подтипа S тот же, что и тип исходного *базового* типа T; здесь базовый как у Temperatures, так и у Headings — тип Integer. Тип определяется *во время компиляции*. Значение подтипа имеет то же самое представление, что и значение базового типа, и допустимо везде, где требуется значение базового типа:

```
Temperature * Compass_Heading
```

это допустимое выражение, но операторы:

```
Temperature := -280;
Compass-Heading := 365;
```

приводят к ошибке, потому что значения выходят за диапазоны подтипов. Нарушения диапазона подтипа выявляются *во время выполнения*.

Подтипы могут быть определены на любом типе, для которого его исходный диапазон может быть разумно ограничен:

```
subtype Upper-Case is Character range 'A'.. 'Z';
U: Upper-Case;
 C: Character;
U := 'a';                    -- Ошибка, выход за диапазон
C := U;                      -- Всегда правильно
U := C;                      -- Может привести к ошибке
```

Подтипы важны для определения массивов, как это будет рассмотрено в разделе 5.4. Кроме того, именованный подтип можно использовать для упрощения многих операторов:

```
if C in Upper-Case then ...          - Проверка диапазона
for C1 in Upper-Case loop ...        — Границы цикла
```

4.6. Производные типы

Вторая интерпретация отношения между двумя аналогичными типами состоит в том, что они представляют разные типы, которые не могут использоваться вместе. В языке Ada такие типы называются *производными (derived) типами* и обозначаются в определении словом new:

```
type Derived_Dharacter is new Character;
C: Character;
D: Derived_Character;
C := D:                              -- Ошибка, типы разные
```

Когда один тип получен из другого типа, называемого *родительским (parent) типом*, он наследует *копию* набора значений и *копию* набора операций, но типы остаются разными. Однако всегда допустимо явное преобразование между типами, полученными друга из друга:

```
D := Derived_Character(C);           -- Преобразование типов
C := Character(D);                   -- Преобразование типов
```

Можно даже задать другое представление для производного типа; преобразование типов будет тогда преобразованием между двумя представлениями (см. раздел 5.8).

Производный тип может включать ограничение на диапазон значений родительского типа:

```
type Upper_Case is new Character range 'A'.. 'Z';
 U: Upper_Case;
 C: Character;
C := Character(U);                    -- Всегда правильно
U := Upper_Case(C);                   -- Может привести к ошибке
```

Производные типы в языке Ada 83 реализуют слабую версию наследования (weak version of inheritance), которая является центральным понятием объектно-ориентированных языков (см. гл. 14). Пересмотренный язык Ada 95 реализует истинное наследование (true inheritance), расширяя понятие производных типов; мы еще вернемся к их изучению.

Целочисленные типы

Предположим, что мы определили следующий тип:

```
type Altitudes is new Integer range 0 .. 60000;
```

Это определение работает правильно, когда мы программируем моделирование полета на 32-разрядной рабочей станции. Что случается, когда мы передадим программу на 16-разрядный контроллер, входящий в состав бортовой электроники нашего самолета? Шестнадцать битов могут представлять целые числа со знаком только до значения 32767. Таким образом, использование производного типа было бы ошибкой (так же, как подтипа или непосредственно Integer) и нарушило бы программную переносимость, которая является основной целью языка Ada.

Чтобы решать эту проблему, можно задать производный целый тип без явного указания базового родительского типа:

```
type Altitudes is range 0 .. 60000;
```

Компилятор должен выбрать представление, которое соответствует требуемому диапазону — integer на 32-разрядном компьютере и Long_integer на 16-разрядном компьютере. Это уникальное свойство позволяет легко писать на языке Ada переносимые программы для компьютеров с различными длинами слова.

Недостаток целочисленных типов состоит в том, что каждое определение создает новый тип, и нельзя писать вычисления, которые используют разные типы без преобразования типа:

```
I: Integer;
A: Altitude;
A := I;                               -- Ошибка, разные типы
A := Altitude(l);                     -- Правильно, преобразование типов
```

Таким образом, существует неизбежный конфликт:

• Подтипы потенциально ненадежны из-за возможности писать смешанные выражения и из-за проблем с переносимостью.

• Производные типы безопасны и переносимы, но могут сделать программу трудной для чтения из-за многочисленных преобразований типов.

4.7. Выражения

Выражение может быть очень простым, состоящим только из литерала (24, V, True) или переменной, но может быть и сложной комбинацией, включающей операции (в том числе вызовы системных или пользовательских функций). В результате *вычисления* выражения получается *значение.*

Выражения могут находиться во многих местах программы: в операторах присваивания, в булевых выражениях условных операторов, в границах for-циклов, параметрах процедур и т. д. Сначала мы обсудим само выражение, а затем операторы присваивания.

Значение литерала — это то, что он обозначает; например, значение 24 — целое число, представляемое строкой битов 0001 1000. Значение переменной V — содержимое ячейки памяти, которую она обозначает. Обратите внимание на возможную путаницу в операторе:

```
V1 :=V2;
```

V2 — *выражение,* значение которого является содержимым некоторой ячейки памяти. V1 — *адрес* ячейки памяти, в которую будет помещено значение V2.

Более сложные выражения содержат функцию с набором параметров или операцию с операндами. Различие, в основном, в синтаксисе: функция с параметрами пишется в префиксной нотации sin (x), тогда как операция с операндами пишется в инфиксной нотации a + b. Поскольку операнды сами могут быть выражениями, можно создавать выражения какой угодно сложности:

```
a + sin(b)*((c-d)/(e+34))
```

В префиксной нотации порядок вычисления точно определен за исключением порядка вычисления параметров отдельной функции:

```
max (sin (cos (x)), cos (sin (y)))
```

Можно написать программы, результат которых зависит от порядка вычисления параметров функции (см. раздел 7.3), но такой зависимости от порядка вычисления следует избегать любой ценой, потому что она является источником скрытых ошибок при переносе программы и даже при ее изменении.

Инфиксной нотации присущи свои проблемы, а именно проблемы старшинства и ассоциативности. Почти все языки программирования придерживаются математического стандарта назначения мультипликативным операциям («*», «/») более высокого старшинства, чем операциям аддитивным («+», «-»), старшинство других операций определяется языком. Крайности реализованы в таких языках, как AP L, в котором старшинство вообще не определено (даже для арифметических операций), и C, где определено 15 уровней старшинства! Частично трудность изучения языка программирования связана с необходимостью привыкнуть к стилю, который следует из правил старшинства.

Примером неинтуитивного назначения старшинства служит язык- Pascal. Булева операция and рассматривается как операция умножения с высоким старшинством, тогда как в большинстве других языков, аналогичных C, ее приоритет ниже, чем у операций отношения. Следующий оператор:

```
if a > b and b > c then ...
```

pascal

является ошибочным, потому что это выражение интерпретируется

```
if a > (b and b) > c then . . .
```

Pascal

и синтаксис оказывается неверен.

Значение инфиксного выражения зависит также от *ассоциативности* операций, т. е. от того, как группируются операции одинакового старшинства: слева направо или справа налево. В большинстве случаев, но не всегда, это не имеет значения (кроме возможного переполнения, как рассмотрено в разделе 4.1). Однако значение выражения, включающего целочисленное деление, может зависеть от ассоциативности из-за усечения:

```
inti=6, j = 7, k = 3;
 i = i * j / k;                    /* результат равен 1 2 или 1 4? */
```

C

В целом, бинарные операции группируются слева направо, так что рассмотренный пример компилируется как:

```
I=(i*j)/k
```

C

в то время как унарные операции группируются справа налево: !++i в языке C вычисляется, как ! (++i).

Все проблемы старшинства и ассоциативности можно легко решить с помощью круглых скобок; их использование ничего не стоит, поэтому применяйте их при малейшем намеке на неоднозначность интерпретации выражения.

В то время как старшинство и ассоциативность определяются языком, порядок вычисления обычно отдается реализаторам для оптимизации. Например, в следующем выражении:

```
(a + Ь) + c + (d + e)
```

не определено, вычисляется a + b раньше или позже d + e, хотя c будет просуммировано с *результатом* a + b раньше, чем *с результатом* d + e. Порядок может играть существенную роль, если выражение вызывает *побочные эффекты*, т. е. если при вычислении подвыражения происходит обращение к функции, которая изменяет глобальную переменную.

Реализация

Реализация выражения, конечно, зависит от реализации операций, используемых в выражении. Однако стоит обсудить некоторые общие принципы.

Выражения вычисляются изнутри наружу; например, a * (b + c) вычисляется так:

```
load R1,b
load R2, c
add R1 , R2          Сложить b и c, результат занести в R1
load R2, a
mult R1.R2           Умножить a на b + c, результат занести в R1
```

Можно написать выражение в форме, которая делает порядок вычисления явным:
явным:

```
 bc + a
```

Читаем слева направо: имя операнда означает загрузку операнда, а знак операции означает применение операции к двум самым последним операндам и замену всех трех (двух операндов и операции) результатом. В этом случае складываются b и c; затем результат умножается на a.

Эта форма называется *польской инверсной записью (reverse polish notation — RPN)* и может использоваться компилятором. Выражение переводится в RPN, и затем компилятор вырабатывает команды для каждого операнда и операции, читая RPN слева направо..

Для более сложного выражения, скажем:

(a + b) * (c + d) * (e + f)

понадобилось бы большее количество регистров для хранения промежуточных результатов: a + b, c + d и т. д. При увеличении сложности регистров не хватит, и компилятору придется выделить неименованные временные пере менные для сохранения промежуточных результатов. Что касается эффектив ности, то до определенной точки увеличение сложности выражения дает лучший результат, чем использование последовательности операторов присваивания, так как позволяет избежать ненужного сохранения промежуточных результатов в памяти. Однако такое улучшение быстро сходит на нет из-за необходимости заводить временные переменные, и в некоторой точке компилятор, возможно, вообще не сможет обработать сложное выражение.

Оптимизирующий компилятор сможет определить, что подвыражение a+b в выражении

(a + b) * c + d * (a + b)

нужно вычислить только один раз, но сомнительно, что он сможет распознать это, если задано

(a + b) * c + d * (b + a)

Если общее подвыражение сложное, возможно, полезнее явно присвоить его переменной, чем полагаться на оптимизатор.

Другой вид оптимизации — *свертка констант.* В выражении:
2.0* 3.14159* Radius

компилятор сделает умножение один раз во время компиляции и сохранит результат. Нет смысла снижать читаемость программы, производя свертку констант вручную, хотя при этом можно дать имя вычисленному значению:

C

```
PI: constants 3.1 41 59;
Two_PI: constant := 2.0 * PI;
Circumference: Float := Two_PI * Radius;
```

4.8. Операторы присваивания

Смысл оператора присваивания:

переменная := выражение;

состоит в том, что значение выражения должно быть помещено по адресу памяти, обозначенному как переменная. Обратите внимание, что левая часть оператора также может быть выражением, если это выражение можно вычислить как адрес:

Ada

```
a(i*(j+1)):=a(i*j);
```

Выражение, которое может появиться в левой части оператора присваивания, называется *l-значением;* константа, конечно, не является 1-значением. Все выражения дают значение и поэтому могут появиться в правой части оператора присваивания; они называются *r-значениями.* В языке обычно не определяется порядок вычисления выражений слева и справа от знака присваивания. Если порядок влияет на результат, программа не будет переносимой.

В языке С само присваивание определено как выражение. Значение конструкции

переменная = выражение;

такое же, как значение выражения в правой части. Таким образом,

C

```
int v1 , v2, v3;
v1 = v2 = v3 = e;
```

означает присвоить (значение) е переменной v3, затем присвоить результат переменной v2, затем присвоить результат переменной v1 и игнорировать конечный результат.

В Ada присваивание является оператором, а не выражением, и многократные присваивания не допускаются. Многократное объявление

V1.V2.V3: Integer :=E;

рассматривается как сокращенная запись для

Ada

```
V1 : Integer :=E;
V2: Integer := E;
V3: Integer := E;
```

а не как многократное присваивание.

Хотя стиль программирования языка С использует тот факт, что присваивание является выражением, этого, вероятно, следует избегать как источник скрытых ошибок программирования. Весьма распространенный класс ошибок вызван тем, что присваивание («=») путают с операцией равенства («==»). В следующем операторе:

```
If (i=j)...
```

C

программист, возможно, хотел просто сравнить i и j, не обратив внимания, что значение i изменяется оператором присваивания. Некоторые С-компиляторы расценивают это как столь плохой стиль программирования, что выдают предупреждающее сообщение.

Полезным свойством языка С является комбинация операции и присваивания:

C

```
v+=e;                    /* Это краткая запись для... */
 v = v + e;              /*    такого оператора. */
```

Операции с присваиванием особенно важны в случае сложной переменной, включающей индексацию массива и т.д. Комбинированная операция не только экономит время набора на клавиатуре, но и позволяет избежать ошибки, если v написано не одинаково с обеих сторон от знака «=». И все же комбинированные присваивания — всего лишь стилистический прием, так как оптимизирующий компилятор может удалить второе вычисление адреса v.

Можно предотвратить присваивание значения объекту, объявляя его как *константу*.

```
const int N = 8;                 /* Константа в языке С */
 N: constant Integer := 8;       — Константа в языке Ada
```

Очевидно, константе должно быть присвоено начальное значение.

Есть различие между константой и *статическим значением (static value)*, которое известно на этапе компиляции:

```
procedure P(C: Character) is
 C1 : constant Character := C;
 C2: constant Character  :='x';
Begin
…
case C is
when C1 =>                -- Ошибка, не статическое значение
when C2 =>                -- Правильно, статическое значение
…
end case;
…
 end P;
```

Ada

Локальная переменная C1 — это постоянный объект, в том смысле что значение не может быть изменено *внутри* процедуры, даже если ее значение будет разным при каждом вызове процедуры. С другой стороны, варианты выбора в case должны быть известны во время компиляции. В отличие от языка C язык C++ рассматривает константы как статические:

```
const int N = 8;
int a[N]; //Правильно в C++, но не в C
```

C++

Реализация

После того как вычислено выражение в правой части присваивания, чтобы сохранить его значение в памяти, нужна как минимум одна команда. Если выражение в левой части сложное (индексация массива и т.д.), то понадобятся дополнительные команды для вычисления нужного адреса памяти.

Если длина значения правой части превышает одно слово, потребуется несколько команд, чтобы сохранить значение в случае, когда компьютер не поддерживает операцию *блочного копирования*, которая позволяет копировать последовательность заданных слов памяти, указав начальный адрес источника, начальный адрес приемника и число копируемых слов.

4.9. Упражнения

1. Прочитайте документацию вашего компилятора и выпишите, какая точность используется для разных целочисленных типов.

2. Запишите 200 + 55 = 255 и 100-150 = -50 в дополнительном коде.

3. Пусть a принимает все значения в диапазонах 50 .. 56 и -56 .. -50, и пусть b равно 7 или -7. Каковы возможные частные q и остатки r при делении a на b? Используйте оба определения остатка (обозначенные rem и mod в Ada) и отобразите результаты в графической форме. Подсказка: если используется rem, r будет иметь знак a; если используется mod, r будет иметь тот же знак, что и b.

4. Что происходит, когда вы выполняете следующую C-программу на компьютере, который сохраняет значения short int в 8 битах, а значения int в 16 битах?

```
short int i: [c]
int j = 280;
for (i = 0; i <j; i++) printf("Hello world");
```

5. Как бы вы реализовали атрибут T'Succ (V) языка Ada, если используется нестандартное представление перечисляемого типа?

6. Что будет печатать следующая программа? Почему?

C

```
int i=2;
int j = 5;
if (i&j)printf("Hello world");
if (i.&&j) printf("Goodbye world");
```

7. Каково значение i после выполнения следующих операторов?

C

```
int | = 0;'
 int a[2] = { 10,11};
i=a[i++];
```

8. Языки C и C++ не имеют операции возведения в степень; почему?

9. Покажите, как могут использоваться модульные типы в Ada 95 и типы целого без знака в C для представления множеств. Насколько переносимым является ваше решение? Сравните с типом множества (set) в языке Pascal.

Глава 5

Составные типы данных

Языки программирования, включая самые первые, поддерживают составные типы данных. С помощью массивов представляются вектора и матрицы, используемые в математических моделях реального мира. Записи используются при обработке коммерческих данных для представления документов различного формата и хранения разнородных данных.

Как и для любого другого типа, для составного типа необходимо описать наборы значений и операций над этими значениями. Кроме того, необходимо решить: как они строятся из элементарных значений, и какие операции можно использовать, чтобы получить доступ к компонентам составного значения? Число встроенных операций над составными типами обычно невелико, поэтому большинство операций нужно явно программировать из операций, допустимых для компонентов составного типа.

Поскольку массивы являются разновидностью записей, мы начнем обсуждение с записей (в языке С они называются *структурами).*

5.1. Записи

Значение типа *запись (record)* состоит из набора значений других типов, называемых *компонентами (components* — Ada), *членами (members* — С) или *полями (fields* —Pascal). При объявлении типа каждое поле получает имя и тип. Следующее объявление в языке С описывает структуру с четырьмя компонентами: одним — типа строка, другим — заданным пользователем перечислением и двумя компонентами целого типа:

```
typedef enum {Black, Blue, Green, Red, White} Colors;

typedef struct {

   char model[20];
   Colors        color;
   int speed;
    int fuel;
} Car_Data;
```

C

Аналогичное объявление в языке Ada таково:

```
type Colors is (Black, Blue, Green, Red, White);

type Car_Data is
        record
           Model: String(1..20);
           Color: Colors:
            Speed: Integer;
            Fuel: Integer;
       end record;
```

Ada

После того как определен тип записи, могут быть объявлены объекты (переменные и константы) этого типа. Между записями одного и того же типа допустимо присваивание:

```
Car_Data c1,c2;
 c1 =c2;
```
C

а в Ada (но не в C) также можно проверить равенство значений этого типа:

```
C1, C2, C3: Car_Data;
 if C1=C2then
C1 =C3;
end if;
```
Ada

Поскольку тип — это набор значений, можно было бы подумать, что всегда можно обозначить* *значение* записи. Удивительно, но этого вообще нельзя сделать; например, язык C допускает значения записи только при инициализации. В Ada, однако, можно сконструировать значение типа *запись*, называемое *агрегатом (aggregate),* просто задавая значение правильного типа для каждого поля. Связь значения с полем может осуществляться по позиции внутри записи или по имени поля:

```
if C1 = (-Peugeot-, Blue, 98, 23) then ...
C1 := (-Peugeot-, Red, C2.Speed, CS.Fuel);
C2 := (Model=>-Peugeot", Speed=>76,
                Fuel=>46, Color=>White);
```
Ada

Это чрезвычайно важно, потому что компилятор выдаст сообщение об ошибке, если вы забудете включить значение для поля; а при использовании отдельных присваиваний легко просто забыть одно из полей:

```
Ada C1.Model :=-Peugeot-;
                                        --Забыли C1.Color
C1.Speed := C2.Speed;
 C1.Fuel := CS.Fuel;
```
Ada

Можно *выбрать* отдельные поля записи, используя точку и имя поля:

```
c1. speed =c1.fuel*x;
```
C

Будучи выбранным, поле записи становится обычной переменной или значением типа поля, и к нему применимы все операции, соответствующие этому типу.

Имена полей записи локализованы внутри определения типа и могут повторно использоваться в других определениях:

```
typedef struct {
            float speed;            /* Повторно используемое имя поля */
      } Performance;
Performance p;
Car_Data c;
p.speed = (float) c.speed;          /* То же самое имя, другое поле*/
```
C

Отдельные записи сами по себе не очень полезны; их значение становится очевидным, только когда они являются частью более сложных структур, таких как массивы записей или динамические структуры, создаваемые с помощью указателей (см. раздел 8.2).

Реализация

Значение записи представляется некоторым числом слов в памяти, достаточным для того, чтобы вместить все поля. На рисунке 5.1 показано размещение записи Car_Data. Поля обычно располагаются в порядке их появления в определении типа записи.

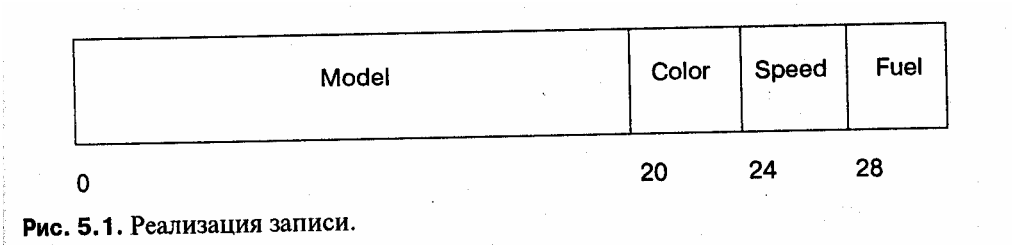

Рис. 5.1. Реализация записи.

Доступ к отдельному полю очень эффективен, потому что величина смещения каждого поля от начала записи постоянна и известна во время компиляции. Большинство компьютеров имеет способы адресации, которые позволяют добавлять константу к адресному регистру при декодировании команды. После того как начальный адрес записи загружен в регистр, для доступа к полям лишние команды уже не нужны:

load R1.&C1 Адрес записи
load R2,20(R1) Загрузить второе поле
load R3,24(R1) Загрузить третье поле

Так как для поля иногда нужен объем памяти, не кратный размеру слова, компилятор может «раздуть» запись так, чтобы каждое поле заведомо находилось на границе слова, поскольку доступ к не выровненному на границу слову гораздо менее эффективен. На 16-разрядном компьютере такое определение типа, как:

```
typedef struct {
char f 1;              /* 1 байт, пропустить 1 байт */
int f2;                /* 2 байта*/
char f3;               /* 1 байт, пропустить 1 байт */
int f4; •              /* 2 байта*/
};
```

C

может привести к выделению четырех слов для каждой записи таким образом, чтобы поля типа int были выровнены на границу слова, в то время как следующие определения:

```
typedef struct { [c]
int f2;                /* 2 байта*/
int  f4;               /* 2 байта*/
charfl ;               /Мбайт*/
char f3;               /* 1 байт */
```

C

потребовали бы только трех слов. При использовании компилятора, который плотно упаковывает поля, можно улучшить эффективность, добавляя фиктивные поля для выхода на границы слова. В разделе 5.8 описаны способы явного распределения полей. В любом случае, *никогда* не привязывайте программу к конкретному формату записи, поскольку это сделает ее непереносимой.

5.2. Массивы

Массив — это запись, все поля которой имеют один и тот же тип. Кроме того, поля (называемые *элементами* или *компонентами)* задаются не именами, а позицией внутри массива. Преимуществом этого типа данных является возможность эффективного доступа к элементу по *индексу*. Поскольку все элементы имеют один и тот же тип, можно вычислить положение отдельного элемента, умножая индекс на размер элемента. Используя индексы, легко найти отдельный элемент массива, отсортировать или как-то иначе реорганизовать элементы.

Индекс в языке Ada может иметь произвольный дискретный тип, т.е. любой тип, на котором допустим «счет». Таковыми являются целочисленные типы и типы перечисления (включая Character и Boolean):

```
type Heat is (Off, Low, Medium, High);
type Temperatures is array(Heat) of Float;
Temp: Temperatures;
```

Ada

Язык С ограничивает индексный тип целыми числами; вы указываете, *сколько* компонентов вам необходимо:

```
#define Max 4
 float temp[Max];
```

C

а индексы неявно изменяются от 0 до числа компонентов без единицы, в данном случае от 0 до 3. Язык С++ разрешает использовать любое константное выражение для задания числа элементов массива, что улучшает читаемость программы:

```
const int last = 3;
 float temp [last+ 1];
```

C++

Компоненты массива могут быть любого типа:

```
typedef struct {... } Car_Data;
Car_Data database [100];
```

C

В языке Ada (но не в С) на массивах можно выполнять операции присваивания и проверки на равенство:

```
type A_Type is array(0..9) of Integer;
A, B, C: AJype;

if A = B then A := C; end if;
```

Ada

Как и в случае с записями, в языке Ada для задания значений массивов, т. е. для агрегатов, предоставляется широкий спектр синтаксических возможностей :

```
A := (1,2,3,4,5,6,7,8,9,10);
A := (0..4 => 1 , 5..9 => 2);          -- Половина единиц, половина двоек
A := (others => 0);                    -- Все нули
```

Ada

В языке С использование агрегатов массивов ограничено заданием начальных значений.

Наиболее важная операция над массивом — индексация, с помощью которой выбирается элемент массива. Индекс, который может быть произвольным выражением индексного типа, пишется после имени массива:

```
type Char_Array is array(Character range 'a'.. 'z') of Boolean;
 A: Char_Array := (others => False);
 C: Character:= 'z';

A(C):=A('a')andA('b');
```

Ada

Другой способ интерпретации массивов состоит в том, чтобы рассматривать их как функцию, преобразующую индексный тип в тип элемента. Язык Ada (подобно языку Fortran, но в отличие от языков Pascal и C) поощряет такую точку зрения, используя одинаковый синтаксис для обращений к функции и для индексации массива. То есть, не посмотрев на объявление, нельзя сказать, является A(1) обращением к функции или операцией индексации массива. Преимущество общего синтаксиса в том, что структура данных может быть первоначально реализована как массив, а позже, если понадобится более сложная структура данных, массив может быть заменен функцией без изменения формы обращения. Квадратные скобки вместо круглых в языках Pascal и C применяются в основном для облегчения работы компилятора.

Записи и массивы могут вкладываться друг в друга в произвольном порядке, что позволяет создавать сложные структуры данных. Для доступа к отдельному компоненту такой структуры выбор поля и индексация элемента должны выполняться по очереди до тех пор, пока не будет достигнут компонент:

```
typedef int A[1 0];              /* Тип массив */
typedef struct {                 /* Тип запись */
A     a;                         /* Массив внутри записи */
char  b;
}   Rec;
Rec   r[10];                     /* Массив записей с массивами типа int внутри */
 int    i,j,k;
k = r[i+l].a[j-1];               /* Индексация, затем выбор поля,затем индексация */
                                 /* Конечный результат — целочисленное значение */
```

C

Обратите внимание, что частичный выбор и индексация в сложной структуре данных дают значение, которое само является массивом или записью:

```
г                Массив записей, содержащих массивы целых чисел
r[i]             Запись, содержащая массив целых чисел
r[i].a           Массив целых чисел
r[i].a[j]        Целое
```

C

и эти значения могут использоваться в операторах присваивания и т.п.

5.3. Массивы и контроль соответствия типов

Возможно, наиболее общая причина труднообнаруживаемых ошибок — это индексация, которая выходит за границы массива:

```
inta[10],
for(i = 0;
i<= 10; i
a[i] = 2*i;
```

C

Цикл будет выполнен и для i = 10, но последним элементом массива является a[9].

Причина распространенности этого типа ошибки в том, что индексные выражения могут быть произвольными, хотя допустимы только индексы, попадающие в диапазон, заданный в объявлении массива. Самая простая ошибка может привести к тому, что индекс получит значение, которое выходит за этот диапазон. Серьезность возникающей ошибки в том, что присваивание a[i] (если i выходит за допустимый диапазон) вызывает изменение некоторой *случайной* ячейки памяти, возможно, даже в области операционной системы. Даже если аппаратная защита допускает изменение данных только в области вашей собственной программы, ошибку будет трудно найти, так как она *проявится* в другом месте, а именно в командах, которые используют измененную память.

Рассмотрим случай, когда числовая ошибка заставляет переменную speed получить значение 20 вместо 30:

C
```
intx=10,y=50;
speed = (x+y)/3;          /*Вычислить среднее! */
```

Проявлением ошибки является неправильное значение speed, и причина (деление на 3 вместо 2) находится здесь же, в команде, которая вычисляет speed. Это проявление непосредственно связано с ошибкой и, используя контрольные точки или точки наблюдения, можно быстро локализовать ошибку. В следующем примере:

C
```
inta[10];
 int speed;
for(i = 0;i<= 10; i ++)
a[i] = 2*j;
```

переменная speed является жертвой того факта, что она была чисто случайно объявлена как раз после a и, таким образом, была изменена совершенно посторонней командой. Вы можете днями прослеживать вычисление speed и не найти ошибку.

Решение подобных проблем состоит в проверке операции индексации над массивами с тем, чтобы гарантировать соблюдение границ. Любая попытка превысить границы массива рассматривается как нарушение контроля соответствия типов. Впервые проверка индексов была предложена в языке Pascal:

pascal
```
type A_Type = array[0..9] of Integer;
A: A_Type;
A[10]:=20;                (*Ошибка*)
```

При контроле соответствия типов ошибка обнаруживается *сразу же*, на своем месте, а не после того, как она «затерла» некоторую «постороннюю» память; целый класс серьезных ошибок исчезает из программ. Точнее, такие ошибки становятся ошибками этапа компиляции, а не ошибками этапа выполнения программы.

Конечно, ничего не дается просто так, и существуют две проблемы контроля соответствия типов для массивов. Первая — увеличение времени выполнения, которое является ценой проверок (мы обсудим это в одном из следующих разделов). Вторая проблема — это противоречие между способом, которым мы работаем с массивами, и способом работы контроля соответствия типов. Рассмотрим следующий пример:

pascal
```
typeA_Type = array[0..9]of Real;          (* Типы массивов *)
type B_Type= array[0..8] of Real;
A: A_Type:                               (* Переменные-массивы *)
B: B_Type;
procedure Sort(var P: A_Type);            (* Параметр-массив *)
sort(A); (* Правильно*) sort(B);          (* Ошибка! *)
```

Два объявления типов определяют два различных типа. Тип фактического параметра процедуры должен соответствовать типу формального параметра, поэтому кажется, что необходимы две разные процедуры Sort, каждая для своего типа. Это не соответствует нашему интуитивному понятию массива и операций над массивом, потому что при тщательном программировании процедур, аналогичных Sort, их делают не зависящими от числа элементов в массиве; границы массива должны быть просто дополнительными параметрами. Обратите внимание, что эта проблема не возникает в языках Fortran или C потому, что в них нет параметров-массивов! Они просто передают адрес начала массива, а программист отвечает за правильное определение и использование границ массива.

В языке Ada изящно решена эта проблема. Тип массива в Ada определяется исключительно *сигнатурой*, т. е. типом индекса и типом элемента. Такой тип называется *типом массива без ограничений*. Чтобы фактически объявить массив, необходимо добавить к типу *ограничение индекса:*

```
type A_Type is array(lnteger range o) of Float;                    Ada

                                    -- Объявление типа массива без ограничений
A: A_Type(0..9);                     — Массив с ограничением индекса
B: A_Type(0..8);                     — Массив с ограничением индекса
```

Сигнатура A_Type — одномерный массив с индексами типа integer и компонентами типа Float; границы индексов *не являются частью* сигнатуры.

Как и в языке Pascal, операции индексации полностью контролируются:

```
A(9) := 20.5;          -- Правильно, индекс изменяется в пределах 0..9     Ada
B(9) := 20.5;          -- Ошибка, индекс изменяется в пределах 0..8
```

Важность неограниченных массивов становится очевидной, когда мы рассматриваем параметры процедуры. Так как тип (неограниченного) массива-параметра определяется только сигнатурой, мы можем вызывать процедуру с любым фактическим параметром этого типа независимо от индексного ограничения:

```
procedure Sort(P: in out A_Type);                                  Ada

                                    — Тип параметра: неограниченный массив
Sort(A);                             -- Типом A является A_Type
Sort(B);                             -- Типом B также является A_Type
```

Теперь возникает вопрос: как процедура Sort может получить доступ к границам массива? В языке Pascal границы были частью типа и таким образом были известны внутри процедуры. В языке Ada ограничения фактического параметра-массива автоматически передаются процедуре во время выполнения и могут быть получены через функции, называемые *атрибутами*. Если A произвольный массив, то:

- A'First — индекс первого элемента A.
- A'Last — индекс последнего элемента A.
- A'Length — число элементов в A.
- A'Range — эквивалент A'First.. A'Last.

Например:

```
procedure Sort(P: in out A_Type) is begin                          Ada
for I in P'Range loop
for J in 1+1 .. P'Lastloop
end Sort;
```

Использование атрибутов массива позволяет программисту писать чрезвычайно устойчивое к изменениям программное обеспечение: любое изменение границ массива автоматически отражается в атрибутах.

Подводя итог, можно сказать: контроль соответствия типов для массивов — мощный инструмент для улучшения надежности программ; однако определение границ массива не должно быть частью статического определения типа.

5.4. Подтипы массивов в языке Ada

Подтипы, которые мы обсуждали в разделе 4.5, определялись добавлением *ограничения диапазона* к дискретному типу (перечисляемому или целочисленному). Точно так же подтип массива может быть объявлен добавлением к типу неограниченного массива *ограничения индекс'*.

```
type A_Type is array(lnteger range o) of Float;
 subtype Line is A_Type(1 ..80);
L, L1, L2: Line;
```

Значение этого именованного подтипа можно использовать как фактический параметр, соответствующий формальному параметру исходного неограниченного типа:

```
Sort(L);
```

В любом случае неограниченный формальный параметр процедуры Sort динамически ограничивается фактическим параметром при каждом вызове процедуры.

Приведенные в разделе 4.5 рассуждения относительно подтипов применимы и здесь. Массивы разных подтипов одного и того же типа могут быть присвоены друг другу (при условии, что они имеют одинаковое число элементов), но массивы разных типов не могут быть присвоены друг другу без явного преобразования типов. Определение именованного подтипа — всего лишь вопрос удобства.

В Ada есть мощные конструкции, называемые *сечениями* (slices) и *сдвигами* (sliding), которые позволяют выполнять присваивания над частями массивов. Оператор

```
L1(10..15):=L2(20..25);
```

присваивает сечение одного массива другому, сдвигая индексы, пока они не придут в соответствие. Сигнатуры типов проверяются во время компиляции, тогда как ограничения проверяются во время выполнения и могут быть динамическими:

```
L1(I..J):=L2(l*K..M+2);
```

Проблемы, связанные с определениями типа для массивов в языке Pascal, заставили разработчиков языка Ada обобщить решение для массивов изящной концепцией подтипов: отделить статическую спецификацию типа от ограничения, которое может быть динамическим.

5.5. Строковый тип

В основном строки — это просто массивы символов, но для удобства программирования необходима дополнительная языковая поддержка. Первое требование: для строк нужен специальный синтаксис, в противном случае работать с массивами символов было бы слишком утомительно. Допустимы оба следующих объявления, но, конечно, первая форма намного удобнее:

```
char s[]= "Hello world";
chars[] = {‘H’,’e’,’l’,’o’,’ ‘,’w’,’o’,’r’,’l’,’d’,’/0’};
```

Затем нужно найти некоторый способ работы с длиной строки. Вышеупомянутый пример уже показывает, что компилятор может определить размер I строки без явного его задания программистом. Язык С использует соглаше-I ние о представлении строк, согласно которому первый обнаруженный нулевой байт завершает строку. Обработка строк в С обычно содержит цикл while вида:

```
while (s[i++]!='\0')... •
```

C

Основной недостаток этого метода состоит в том, что если завершающий ноль почему-либо отсутствует, то память может быть затерта, так же как и при любом выходе за границы массива:

```
char s[11]= "Hello world";          /* Не предусмотрено место
                                    для нулевого байта*/
chart[11];
 strcpy(t, s);                      /* Копировать set. Какой длины s? */
```

C

Другие недостатки этого метода:

• Строковые операции требуют динамического выделения и освобождения памяти, которые относительно неэффективны.

• Обращения к библиотечным строковым функциям приводят к повторным вычислениям длин строк.

• Нулевой байт не может быть частью строки.

Альтернативное решение, используемое некоторыми диалектами языка Pascal, состоит в том, чтобы включить явный байт длины как неявный нулевой символ строки, чья максимальная длина определяется при объявлении:

```
S:String[10];
S := 'Hello world';          (* Требуется 11 байтов *)
writeln(S);
S:='Hello';
writeln(S);
```

Pascal

Сначала программа выведет «Hello worl», так как строка будет усечена до объявленной длины. Затем выведет «Hello», поскольку writeln принимает во внимание неявную длину. К сожалению, это решение также небезупречно, потому что возможно непосредственное обращение к скрытому байту длины и затирание памяти:

```
s[0]:=15;
```

Pascal

В Ada есть встроенный тип неограниченного массива, называемый String, со следующим определением:

```
type String is array(Positive range <>) of Character;
```
Ada

Каждая строка должна быть фиксированной длины и объявлена с индексным ограничением:

Ada

```
S:String(1..80);
```

В отличие от языка C, где вся обработка строк выполняется с использованием библиотечных процедур, подобных strcpy, в языке Ada над строками допускаются такие операции, как конкатенация «&», равенство и операции отношения, подобные «<». Поскольку строго предписан контроль соответствия типов, нужно немного потренироваться с атрибутами, чтобы заставить все заработать:

```
S1: constant String := "Hello";
S2: constant String := "world";
T: String(1  .. S1 'Length + 1 + S2'Length) := S1 & ' ' & S2;
Put(T);                                    -- Напечатает Hello world
```
Ada

Точная длина T должна быть вычислена до того, как выполнится присваивание! К счастью, Ada поддерживает атрибуты массива и конструкцию для создания подмассивов (называемых сечениями — slices), которые позволяют выполнять такие вычисления переносимым способом.

Ada 83 предоставляет базисные средства для определения строк нефиксированной длины, но не предлагает необходимых библиотечных подпрограмм для обработки строк. Чтобы улучшить переносимость, в Ada 95 определены стандартные библиотеки для всех трех категорий строк: фиксированных, изменяемых (как в языке Pascal) и динамических (как в C).

5.6. Многомерные массивы

Многомерные матрицы широко используются в математических моделях физического мира, и многомерные массивы появились в языках программирования начиная с языка Fortran. Фактически есть два способа определения многомерных массивов: прямой и в качестве сложной структуры. Мы ограничимся обсуждением двумерных массивов; обобщение для большей размерности делается аналогично.

Прямое определение двумерного массива в языке Ada можно дать, указав два индексных типа, разделяемых запятой:

```
type Two is
array(Character range <>, Integer range <>) of Integer;
 T:Two('A'..'Z', 1 ..10); I: Integer;
C: Character;
T('XM*3):=T(C,6);
```
Ada

Как показывает пример, две размерности не обязательно должны быть одного и того же типа. Элемент массива выбирают, задавая оба индекса.

Второй метод определения двумерного массива состоит в том, чтобы определить тип, который является массивом массивов:

Ada

```
type l_Array is array( 1.. 10) of Integer;
type Array_of_Array is array (Character range <>) of l_Array;
T:Array_of_Array('A'..'Z');

I: Integer;
C: Character;

T('X')(I*3):=T(C)(6);
```

Преимущество этого метода в том, что можно получить доступ к элементам второй размерности (которые сами являются массивами), используя одну операцию индексации:

Ada

```
T('X') :=T('Y');                    -- Присвоить массив из 10 элементов
```

Недостаток же в том, что для элементов второй размерности должны быть заданы ограничения до того, как эти элементы будут использоваться для определения первой размерности.

В языке С доступен только второй метод и, конечно, только для целочисленных индексов:

C

```
inta[10][20];
 a[1] = a[2];                       /* Присвоить массив из 20 элементов */
```

Язык Pascal не делает различий между двумерным массивом и массивом массивов; так как границы считаются частью типа массива, это не вызывает никаких проблем.

5.7. Реализация массивов

При реализации элементы массива размещаются в памяти последовательно. Если задан массив А, то адрес его элемента A(l) есть (см. рис. 5.2.):

addr (A) + *size (element)* * (/ - *A.'First)*

Например: адрес A(4) равен 20 + 4 * (4 - 1) = 32.

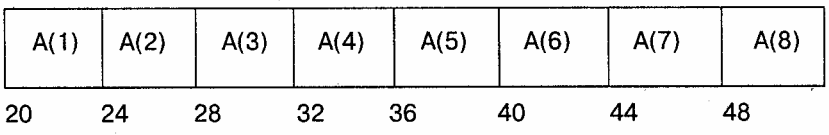

Рис. 5.2. Размещение элементов массива.

Сгенерированный машинный код будет выглядеть так:

```
L
oad      R1,l              Получить индекс
sub      R1,A'First        Вычесть нижнюю границу
multi    R1 ,size          Умножить на размер — > смещение
add      R1 ,&A            Добавить адрес массива — > адрес элемента
load     R2,(R1)           Загрузить содержимое
```

Вы, возможно, удивитесь, узнав, что для каждого доступа к массиву нужно столько команд! Существует много вариантов оптимизации, которые могут улучшить этот код. Сначала отметим, что если A'First — ноль, то нам не нужно вычитать индекс первого элемента; это объясняет, почему разработчики языка С сделали так, что индексы всегда начинаются с

нуля. Даже если A'First — не ноль, но известен на этапе компиляции, можно преобразовать вычисление адреса следующим образом:

*(addr (A) - size (element) * A'First) + (size (element)* i)

Первое выражение в круглых скобках можно вычислить при компиляции, экономя на вычитании во время выполнения. Это выражение будет известно во время компиляции при обычных обращениях к массиву:

```
A:A_Type(1..10);
A(I):=A(J);
```
Ada

но не в том случае, когда массив является параметром:

```
procedure Sort(A: A_Type) is
 begin
…
A(A'First+1):=A(J);
…
 end Sort;
```
Ada

Основное препятствие для эффективных операций с массивом — умножение на размер элемента массива. К счастью, большинство массивов имеют простые типы данных, такие как символы или целые числа, и размеры их элементов представляют собой степень двойки. В этом случае дорогостоящая операция умножения может быть заменена эффективным сдвигом, так как сдвиг влево на n эквивалентен умножению на 2". В случае массива записей можно повысить эффективность (за счет дополнительной памяти), дополняя записи так, чтобы их размер был кратен степени двойки. Обратите внимание, что на переносимость программы это не влияет, но само *улучшение* эффективности не является переносимым: другой компилятор может скомпоновать запись по-другому.

Программисты, работающие на C, могут иногда повышать эффективность обработки массивов, явно программируя доступ к элементам массива с помощью указателей вместо индексов. Следующие определения:

```
typedef struct {
…
int field;
 } Rec;
Rec a[100];
```
C

могут оказаться более эффективными (в зависимости от качества оптимизаций в компиляторе) при обращении к элементам массива по указателю:

```
Rec* ptr;

for (ptr = &a; ptr < &a+100*sizeof(Rec); ptr += sizeof(Rec))
 ...ptr-> field...;
```
C

чем при помощи индексирования:

```
for(i=0; i<100;i++)
…a[i].field…
```

Однако такой стиль программирования чреват множеством ошибок; кроме того, такие программы тяжело читать, поэтому его следует применять только в исключительных случаях.

В языке С возможен и такой способ копирования строк:

C

```
while (*s1++ = *s2++)
```

в котором перед точкой с запятой стоит пустой оператор. Если компьютер поддерживает команды блочного копирования, которые перемещают содержимое блока ячеек памяти по другому адресу, то эффективнее будет язык типа Ada, который допускает присваивание массива. Вообще, тем, кто программирует на С, следует использовать библиотечные функции, которые, скорее всего, реализованы более эффективно, чем примитивный способ, показанный выше.

Многомерные массивы могут быть очень неэффективными, потому что каждая лишняя размерность требует дополнительного умножения при вычислении индекса. При работе с многомерными массивами нужно также понимать, как размещены данные. За исключением языка Fortran, все языки хранят двумерные массивы как последовательности строк. Размещение

Ada

```
type T is array( 1 ..3, 1 ..5) of Integer;
```

показано на рис. 5.3. Такое размещение вполне естественно, поскольку сохраняет идентичность двумерного массива и массива массивов. Если в вычислении перебираются все элементы двумерного массива, проследите, чтобы последний индекс продвигался во внутреннем цикле:

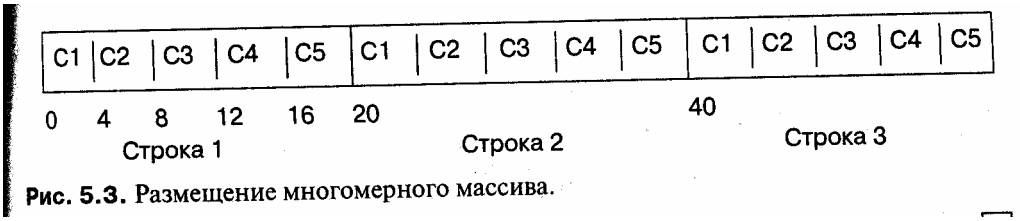

Рис. 5.3. Размещение многомерного массива.

C

```
intmatrix[100][200];

for(i = 0;i<100;i++)
for (j = 0; j < 200; j++)
m[i][j]=…;
```

Причина в том, что операционные системы, использующие разбиение на страницы, работают намного эффективнее, когда адреса, по которым происходят обращения, находятся близко друг к другу.

Если вы хотите выжать из С-программы максимальную производительность, можно игнорировать двумерную структуру массива и имитировать одномерный массив:

C

```
for (i=0; i< 1 00*200; i++)
m[]0[i]=…;
```

Само собой разумеется, что применять такие приемы не рекомендуется, а в случае использования их следует тщательно задокументировать.

Контроль соответствия типов для массива требует, чтобы попадание индекса в границы проверялось перед каждым доступом к массиву. Издержки этой проверки велики: два сравнения и переходы. Компиляторам для языков типа Ada приходится проделывать значительную работу, чтобы оптимизировать команды обработки массива. Основной технический прием — использование доступной информации. В следующем примере:

```
for I in A' Range loop
if A(I) = Key then ...
```
Ada

индекс I примет только допустимые для массива значения, так что никакая проверка не нужна. Вообще, оптимизатор лучше всего будет работать, если все переменные объявлены с максимально жесткими ограничениями.

Когда массивы передаются как параметры на языке с контролем соответствия типов:

```
type A_Type is array(lnteger range o) of Integer;
procedure Sort(A: A_Type) is ...
```
Ada

границы также неявно должны передаваться в структуре данных, называемой *дескриптором массива (dope vector)* (рис. 5.4). Дескриптор массива содержит

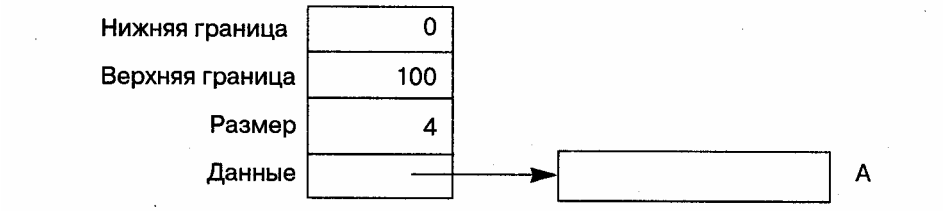

Рис. 5.4. Дескриптор массива.

верхнюю и нижнюю границы, размер элемента и адрес начала массива. Как мы видели, это именно та информация, которая нужна для вычисления адресов при индексации массива.

5.8. Спецификация представления

В этой книге неоднократно подчеркивается значение интерпретации программы как абстрактной модели реального мира. Однако для таких программ, как операционные системы, коммуникационные пакеты и встроенное программное обеспечение, необходимо манипулировать данными на физическом уровне их представления в памяти.

Вычисления над битами

В языке С есть булевы операции, которые выполняются побитно над значениями *целочисленных типов:* «&» (and), «|» (or), «л» (xor), «~» (not).

Булевы операции в Ada — and, or, xor, not — также могут применяться к булевым массивам:

```
type Bool_Array is array(0..31) of Boolean;
B1: Bool_Array:=(0..15=>True, 16..31 => False);
B2: Bool_Array := (0..15 => False, 16..31 => True);
B1 :=B1 orB2;
```
Ada

Однако само объявление булевых массивов не гарантирует, что они представляются как битовые строки; фактически, булево значение обычно представляется как целое число. Добавление управляющей команды

```
pragma Pack(Bool_Array);
```
Ada

требует, чтобы компилятор упаковывал значения массива как можно плотнее. Поскольку для булева значения необходим только один бит, 32 элемента массива могут храниться в 32-

разрядном слове. Хотя таким способом и обеспечиваются требуемые функциональные возможности, однако гибкости, свойственной языку С, достичь не удастся, в частности, из-за невозможности использовать в булевых вычислениях такие восьмеричные или шестнадцатеричные константы, как OxfOOf OffO. Язык Ada обеспечивает запись для таких констант, но они являются целочисленными значениями, а не булевыми массивами, и поэтому не могут использоваться в поразрядных вычислениях.

Эти проблемы решены в языке Ada 95: в нем для поразрядных вычислений могут использоваться модульные типы (см. раздел 4.1):

```
type Unsigned_Byte is mod 256;
UI,U2: Unsigned_Byte;
```
Ada

```
U1 :=U1 andU2;
```

Поля внутри слов

Аппаратные регистры обычно состоят из нескольких полей. Традиционно доступ к таким полям осуществляется с помощью сдвига и маскирования; оператор

```
field = (i » 4) & 0x7;
```

извлекает трехбитовое поле, находящееся в четырех битах от правого края слова i. Такой стиль программирования опасен, потому что очень просто сделать ошибку в числе сдвигов и в маске. Кроме того, при малейшем изменении размещения полей может потребоваться значительное изменение программы.

- Изящное решение этой проблемы впервые было сделано в языке Pascal: использовать обычные записи, но упаковывать несколько полей в одно слово. Обычный доступ к полю Rec.Field автоматически переводится компилятором в правильные сдвиг и маску.

В языке Pascal размещение полей в слове явно не задается; в других языках такое размещение можно описать явно. Язык С допускает спецификаторы разрядов в поле структуры (при условии, что поля имеют целочисленный тип):

```
typedef struct {
int : 3;                    /* Заполнитель */
 int    f1    :1;
 int    f2    :2;
```
C

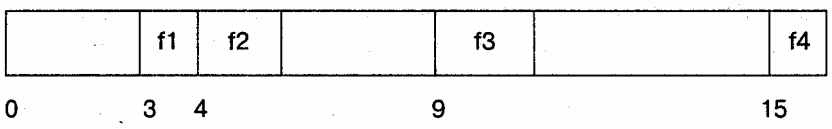

Рис. 5.5. Поля записи внутри слова.

```
int : 3;                    /* Заполнитель */
int    f3    :2;
int : 4;                    /* Заполнитель */
int    f4    :1;
}reg;
```
C

и это позволяет программисту использовать обычную форму предложений присваивания (хотя поля и являются частью слова), а компилятору реализовать эти присваивания с помощью сдвигов и масок:

```
reg r;
[c] int i;
i = r.f2;
r.f3 = i;
```

C

Язык Ada неуклонно следует принципу: объявления типа должны быть абстрактными. В связи с этим *спецификации представления (representation specifications)* используют свою нотацию и пишутся отдельно от объявления типа. К следующим ниже объявлениям типа:
type Heat is (Off, Low, Medium, High);

```
type Reg is
     record
F1: Boolean;
F2: Heat;
F3: Heat;
F4: Boolean;
end record;
```

Ada

может быть добавлена такая спецификация:

```
for Reg use
 record
F1 at 0 range 3..3;
 F2 at Orange 4..5;
F3at 1 range 1..2;
F4at 1 range 7..7;
end record;
```

Ada

Конструкция at определяет байт внутри записи, а range определяет отводимый полю диапазон разрядов, причем мы знаем, что достаточно одного бита для значения Boolean и двух битов для значения Heat. Обратите внимание, что заполнители не нужны, потому что определены точные позиции полей.

Если разрядные спецификаторы в языке C и спецификаторы представления в Ada правильно запрограммированы, то обеспечена безошибочность всех последующих обращений.

Порядок байтов в числах

Как правило, адреса памяти растут начиная с нуля. К сожалению, архитектуры компьютеров отличаются способом хранения в памяти многобайтовых значений. Предположим, что можно независимо адресовать каждый байт и что каждое слово состоит из четырех байтов. В каком виде будет храниться целое число 0x04030201: начиная *со старшего конца (big endian)*, т. е. так, что старший байт имеет меньший адрес, или начиная *с младшего конца (little endian)*, т. е. так, что младший байт имеет меньший адрес? На рис. 5.6 показано размещение байтов для двух вариантов.

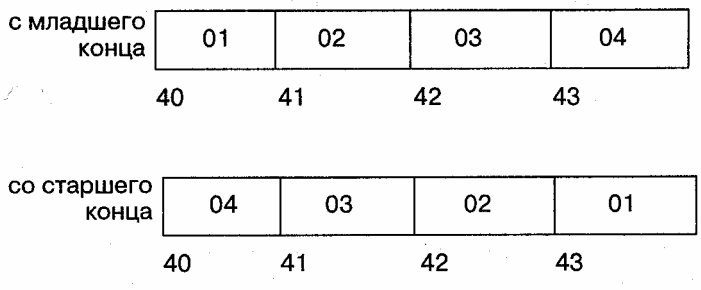

Рис. 5.6. Два способа размещения байтов числа в слове.

В компиляторах такие архитектурные особенности компьютеров, естественно, учтены и полностью прозрачны (невидимы) для программиста, если он описывает свои данные на должном уровне абстракции.

Однако при использовании спецификаций представления разница между двумя соглашениями может сделать программу непереносимой. В языке Ada 95 порядок битов слова может быть задан программистом, так что для переноса программы, использующей спецификации представления, достаточно заменить всего лишь спецификации.

Производные типы и спецификации представления в языке Ada

Производный тип в языке Ada (раздел 4.6) определен как новый тип, чьи значения и операции такие же, как у родительского типа. Производный тип может иметь представление, отличающееся от родительского типа. Например, если определен обычный тип Unpacked_Register:

```
type Unpacked_Register is
 record
…
end record;
```

Ada

можно получить новый тип и задать спецификацию представления, связанную с производным типом:

```
type Packed_Register is new Unpacked_Register;
for Packed_Register use
 record
…
end record;
```

Ada

Преобразование типов (которое допустимо между любыми типами, полученными друг из друга) вызывает изменение представления, а именно упаковку и распаковку полей слов в обычные переменные:

```
U: Unpacked_Register;
P: Packed_Register;

U := Unpacked_Register(P);
P := Packed_Register(U);
```

Ada

Это средство может сделать программы более надежными, потому что, коль скоро написаны правильные спецификации представления, остальная часть программы становится полностью абстрактной.

5.9. Упражнения

1. Упаковывает ваш компилятор поля записи или выравнивает их на границы слова?

2. Поддерживает ли ваш компьютер команду блочного копирования, и использует ли ее ваш компилятор для операций *присваивания над массивами* и записями?

3. Pascal содержит конструкцию with, которая открывает область видимости имен так, что имена полей записи можно использовать непосредственно:

```
type Rec =
     record
          Field 1: Integer;
          Field2: Integer;
     end;
 R: Rec;
```

Paskal

```
with R do Field 1 := Field2;          (* Правильно, непосредственная видимость *)
```

Каковы преимущества и недостатки этой конструкции? Изучите в Ada конструкцию renames и покажите, как можно получить некоторые аналогичные функциональные возможности. Сравните две конструкции.

4. Объясните сообщение об ошибке, которое вы получаете в языке C при попытке присвоить один массив другому:

```
inta1[10],a2[10]:
a1 =a2;
```

C

5. Напишите процедуры sort на языках Ada и C и сравните их. Убедитесь, что вы используете атрибуты в процедуре Ada так, что процедура будет обрабатывать массивы с произвольными индексами.

6. Как оптимизирует ваш компилятор операции индексации массива?

7. В языке Icon имеются ассоциативные массивы, называемые таблицами, в которых строка может использоваться как индекс массива:

```
count["begin"] = 8;
```

Реализуйте ассоциативные массивы на языках Ada или C.

8. Являются следующие два типа одним и тем же?

Ada

```
type Array_Type_1 is array(1 ..100) of Float;
type Array_Type_2 is array(1 ..100) of Float;
```

Языки Ada и C++ используют *эквивалентность имен:* каждое объявление типа объявляет новый тип, так что будут объявлены два типа. При *структурной эквивалентности* (используемой в языке Algol 68) объявления типа, которые выглядят одинаково, определяют один и тот же тип. Каковы преимущества и недостатки этих двух подходов?

9. В Ada может быть определен массив анонимного типа. Допустимо ли присваивание в следующем примере? Почему?

```
A1, A2: аггау( 1.. 10) of Integer;
 A1 :=A2;
```

Ada

Глава 6

Управляющие структуры

Управляющие операторы предназначены для изменения порядка выполнения команд программы. Есть два класса хорошо структурированных управляющих операторов: операторы выбора (if и case), которые выбирают одну из двух или нескольких возможных альтернативных последовательностей выполнения, и операторы цикла (for и while), которые многократно выполняют последовательность операторов.

6.1. Операторы switch и case

Оператор выбора используется для выбора одного из нескольких возможных путей, по которому должно выполняться вычисление (рис. 6.1). Обобщенный оператор выбора называется switch-оператором в языке C и case-оператором в других языках.

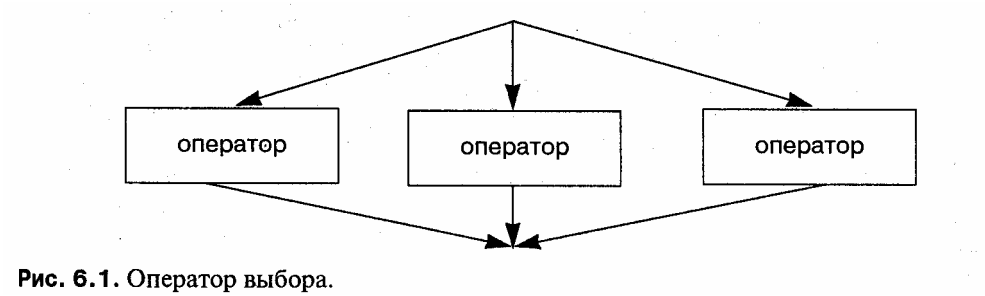

Рис. 6.1. Оператор выбора.

Switch-оператор состоит из выражения (expression) и оператора (statement) для каждого возможного значения (value) выражения:

```
switch (expression) {
case value_1:
 statement_1;
    break;
case value_2:
statement_2;
break;
….
}
```

C

Выражение вычисляется, и его результат используется для выбора оператора, который будет выполнен; на рис. 6. 1 выбранный оператор представляет путь. Отсюда следует, что для каждого возможного значения выражения должна существовать *в точности* одна case-альтернатива. Для целочисленного выражения это невозможно, так как нереально написать свой оператор для каждого 32-разрядного целочисленного значения. В языке Pascal case-оператор используется только для типов, которые имеют небольшое число значений, тогда как языки C и Ada допускают альтернативу по умолчанию (default), что позволяет использовать case-оператор даже для таких типов, как Character, которые имеют сотни значений:

```
default:
 default_statement;
 break;
```
C

,

Если вычисленного значения выражения не оказывается в списке, то выполняется оператор, заданный по умолчанию (default_statement). В языке С, если альтернатива default отсутствует, по умолчанию подразумевается пустой оператор. Эту возможность использовать не следует, потому что читатель программы не может узнать, подразумевался ли пустой default-оператор, или программист просто забыл задать необходимые операторы.

Во многих случаях операторы для двух или нескольких альтернатив идентичны. В языке С нет специальных средств для этого случая (см. ниже); а в Ada есть обширный набор синтаксических конструкций Для группировки альтернатив:

```
C: Character;
case C is
when 'A'.. 'Z'                  => statement_1;
when '0'.. '9'                  => statement_2;
when '+' | '-' |' *' | '/'      =>statement_3;
when others                      => statement_4;
end case;
```
Ada

В Ada альтернативы представляются зарезервированным ключевым словом when, а альтернатива по умолчанию называется others. Case-альтернативаможет содержать диапазон значений value_1 .. value_2 или набор значений, разделенных знаком «|».

Оператор break в языке С

В языке С нужно явно завершать каждую case-альтернативу оператором break, иначе после него вычисление «провалится» на следующую case-альтернативу. Можно воспользоваться такими «провалами» и построить конструкцию, напоминающую многоальтернативную конструкцию языка Ada:

```
 char c;
switch (c) {
        case 'A': case'B': ... case'Z':
        statement_1 ;
        break;
   case'O': ... case '9':
   statement_2;
   break;
        case '+'; case '-': case '*': case '/':
        statement_3 :
         break;
 default:
statement_4;
 break;
```
C

Поскольку каждое значение должно быть явно написано, switch-оператор в языке С далеко не так удобен, как case-оператор в Ada.

В обычном программировании «провалы» использовать не стоит:

```
switch (e) {
    casevalue_1:
      statement_1 ;                  /* После оператора statemerrM */
  case value_2:
  statement_2;                    /* автоматический провал на  statement_2. */
break;
}
```

C

Согласно рис. 6.1 switch -оператор должен использоваться для выбора одного из нескольких возможных путей. «Провал» вносит путаницу, потому что при достижении конца пути управление как бы возвращается обратно к началу дерева выбора. Кроме того, с точки зрения семантики не должна иметь никакого значения последовательность, в которой записаны варианты выбора (хотя в смысле эффективности порядок может быть важен). При сопровождении программы нужно иметь возможность свободно изменять существующие варианты выбора или вставлять новые варианты, не опасаясь внести ошибку. Такую программу, к тому же, трудно тестировать и отлаживать: если ошибка прослежена до оператора statement_2, трудно узнать, был оператор достигнут непосредственным выбором или в результате провала. Чем пользоваться «провалом», лучше общую часть (common_code) оформить как процедуру:

```
switch (e) {
    case value_1 :
    statement_1 ;
    common_code();
    break;
 case value_2:
common_code();
break;
}
```

C

Реализация

Самым простым способом является компиляция case-оператора как последовательности проверок:

```
compute          R1 ,expr          Вычислить выражение
jump_eq          R1,#value_1,L1
jump_eq          R1,#value_2 ,L2
…                                  Другие значения
default_statement                  Команды, выполняемые по
                                   умолчанию
 jump            End_Case


L1:  statement_1                   Команды для statement_1
     jump        End_Case

L2:  statement_2                   Команды для statement_2
     jump         End_Case
…                                  Команды для других операторов
End_Case:
```

С точки зрения эффективности очевидно, что чем ближе к верхней части оператора располагается альтернатива, тем более эффективен ее выбор; вы можете переупорядочить альтернативы, чтобы извлечь пользу из этого факта (при условии, что вы не используете «провалы»!).

Некоторые case-операторы можно оптимизировать, используя таблицы переходов. Если набор значений выражения образует короткую непрерывную последовательность, то можно использовать следующий код (подразумевается, что выражение может принимать значения от 0 до 3):

```
                compute     R1,expr
                mult        R1,#len_of_addr     expr* длина_адреса
                add         R1 ,&table          + адрес_начала_таблицы
                jump        (R1)                Перейти по адресу в регистре R1

table:                                          Таблица переходов
                addr(L1)
                addr(L2)
                addr(L3)
                addr(L4)

L1:             statement_1
                jump            End_Case
L2:             statement_2
                jump            End_Case
L3:             statement_3
                jump            End_Case
L4:             statement_4
End_Case:
```

Значение выражения используется как индекс для таблицы адресов операторов, а команда jump осуществляет переход по адресу, содержащемуся в регистре. Затраты на реализацию варианта с таблицей переходов фиксированы и невелики для *всех* альтернатив.

Значение выражения обязательно должно лежать внутри ожидаемого диапазона (здесь от 0 до 3), иначе будет вычислен недопустимый адрес, и произойдет переход в такое место памяти, где может даже не быть выполнимой команды! В языке Ada выражение часто может быть проверено во время компиляции:

Ada

```
type Status is (Off, WarmJJp, On, Automatic);
S: Status;
case S is ...                       -- Имеется в точности четыре значения
```

В других случаях будет необходима динамическая проверка, чтобы гарантировать, что значение лежит внутри диапазона. Таблицы переходов совместимы даже с альтернативой по умолчанию при условии, что явно заданные варианты выбора расположены непрерывно друг за другом. Компилятор просто вставляет динамическую проверку допустимости использования таблицы переходов; при отрицательном результате проверки вычисляется альтернатива по умолчанию.

Выбор реализации обычно оставляется компилятору, и нет никакой возможности узнать, какая именно реализация используется, без изучения машинного кода. Из документации оптимизирующего компилятора вы, возможно, и узнаете, при каких условиях будет компилироваться таблица переходов. Но даже если вы учтете их при программировании, ваша программа не перестанет быть переносимой, потому что сам case-оператор — переносимый; однако разные компиляторы могут реализовывать его по-разному, поэтому увеличение эффективности не является переносимым.

6.2. Условные операторы

Условный оператор — это частный случай case- или switch-оператора, в котором выражение имеет булев тип. Так как булевы типы имеют только два допустимых значения, условный оператор делает выбор между двумя возможными путями. Условные операторы — это, вероятно, наиболее часто используемые управляющие структуры, поскольку часто применяемые операции отношения возвращают значения булева типа:

```
if (x > y)
        statement_1;                                    C
else
       statement_2;
```

Как мы обсуждали в разделе 4.4, в языке C нет булева типа. Вместо этого применяются целочисленные значения с условием, что ноль это «ложь» (False), а не ноль — «истина» (True).

Распространенная ошибка состоит в использовании условного оператора для создания булева значения:

```
if X > Y then
         Result = True;                                 Ada
else
          Result = False;
end if;
```

вместо простого оператора присваивания:

```
 Result := X > Y;                                       Ada
```

Запомните, что значения и переменные булева типа являются «полноправными» объектами: в языке C они просто целые, а в Ada они имеют свой тип, но никак не отличаются от любого другого типа перечисления. Тот факт, что булевы типы имеют специальный статус в условных операторах, не накладывает на них никаких ограничений.

Вложенные if-операторы

Альтернативы в if-операторе сами являются операторами; в частности, они могут быть и if-операторами:

```
if(x1>y1)
     if (x2 > y2)
           statement_1;                                 C
    else
           statement_2;
else
       if (x3 > y3)
            statemen_3;
       else
            statement_4;
```

Желательно не делать слишком глубоких вложений управляющих структур (особенно if-операторов) — максимум три или четыре уровня. Причина в том, что иначе становится трудно проследить логику различных путей. Кроме того, структурирование исходного текста с помощью отступов — всего лишь ориентир: если вы пропустите else, синтаксически оператор может все еще оставаться правильным, хотя работать он будет неправильно.

Другая возможная проблема — «повисший» else:

```
if (x1 > y1)
  if (x2 > y2)
        statement_1;
  else
        statement_2;
```

C

Как показывают отступы, определение языка связывает else с наиболее глубоко вложенным if-оператором. Если вы хотите связать его с внешним if-оператором, нужно использовать скобки:

```
if(x1>y1){
    if (x2 > y2)
       statement_1; }
else
statement_2;
```

Вложенные if-операторы могут определять полное двоичное дерево выборов (рис. 6.2а) или любое произвольное поддерево. Во многих случаях тем не менее необходимо выбрать одну из последовательностей выходов (рис. 6.26).

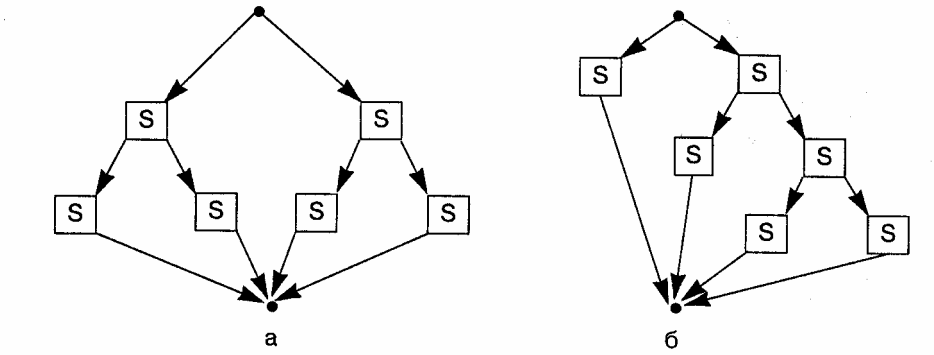

Рис. 6.2. if-операторы: а — полностью вложенный, б — последовательный.

Если выбор делается на основе выражения, можно воспользоваться switch-оператором. Однако, если выбор делается на основе последовательности выражений отношения, понадобится последовательность вложенных if-опера-торов. В этом случае принято отступов не делать:

```
if (x > y) {
…
 } else if (x > z) {
} else if(y < z) {
} else {
...
}
```

C

Явный end if

Синтаксис if-оператора в языке C (и Pascal) требует, чтобы каждый вариант выбора был одиночным оператором. Если вариант состоит из нескольких операторов, они должны быть объединены в отдельный *составной (compound)* оператор с помощью скобок ({,} в языке C и begin, end в Pascal). Проблема такого синтаксиса состоит в том, что если закрывающая скобка пропущена, то компиляция будет продолжена без извещения об ошибке в том месте, где она сделана. В лучшем случае отсутствие скобки будет отмечено в конце компиляции; а в худшем — количество скобок сбалансируется пропуском какой-либо *открывающей* скобки и ошибка станет скрытой ошибкой этапа выполнения.

Эту проблему можно облегчить, явно завершая if-оператор. Пропуск закрывающей скобки будет отмечен сразу же, как только другая конструкция (цикл или процедура) окажется завершенной другой скобкой. Синтаксис if-оператора языка Ada таков:

```
if expression then
   statement_list_1;
else                                                    Ada
    statement_list_2;
end if;
```

Недостаток этой конструкции в том, что в случае последовательности условий (рис. 6.26) получается запутанная последовательность из end if. Чтобы этого избежать, используется специальная конструкция elsif, которая представляет другое условие и оператор, но не другой if-оператор, так что не требуется никакого дополнительного завершения:

```
if x > y then
….
elsif x >z then                                         Ada
….
elsif y > z then
…
else
…
end if;
```

Реализация

Реализация if-оператора проста:

```
        compute     R1,expression
        jump_eq     R1,L1           False представляется как 0
        statement_1
        jump        L2
L1:     statement_2
L2:
```

Обратите внимание, что вариант False немного эффективнее, чем вариант True, так как последний выполняет лишнюю команду перехода. На первый взгляд может показаться, что условие вида:

```
                                                        C
   if (!expression)
```

потребует дополнительную команду для отрицания значения. Однако компиляторы достаточно интеллектуальны для того, чтобы заменить изначальную команду jump_false на jump_true.

Укороченное и полное вычисления

Предположим, что в условном операторе не простое выражение отношения, а составное:

```
if (x > y) and (y > z) and (z < 57) then...
```
Ada

Есть два способа реализации этого оператора. Первый, называемый *полным вычислением*, вычисляет каждый из компонентов, затем берет булево произведение компонентов и делает переход согласно полученному результату. Вторая реализация, называемая *укороченным вычислением (short-circuit)**, вычисляет компоненты один за другим: как только попадется компонент со значением False, делается переход к False-варианту, так как все выражение, очевидно, имеет значение False. Аналогичная ситуация происходит, если составное выражение является or-выражением: если какой-либо компонент имеет значение True, то, очевидно, значение всего выражения будет True.

Выбор между двумя реализациями обычно может быть предоставлен компилятору. В целом укороченное вычисление требует выполнения меньшего числа команд. Однако эти команды включают много переходов, и, возможно, на компьютере с большим кэшем команд (см. раздел 1.7) эффективнее вычислить все компоненты, а переход делать только после полного вычисления.

В языке Pascal оговорено полное вычисление, потому что первоначально он предназначался для компьютера с большим кэшем. Другие языки имеют два набора операций: один для полного вычисления булевых значений и другой — для укороченного. Например, в Ada and используется для полностью вычисляемых булевых операций на булевых и модульных типах, в то время как and then определяет укороченное вычисление:

```
if (x > y) and then (y > z) and then (z < 57) then...
```
Ada

Точно так же or else — эквивалент укороченного вычисления для or.

Язык С содержит три логических оператора: «!» (не), « &&» (и), и «||» (или). Поскольку в С нет настоящего типа Boolean, эти операторы работают с целочисленными операндами и результат определяется в соответствии с интерпретацией, описанной в разделе 4.4. Например, a && b равно единице, если оба операнда не нулевые. Как «&&», так и «||» используют укороченное вычисление. Убедитесь, что вы не спутали эти операции с поразрядными операциями (раздел 5.8).

Относительно стиля программирования можно сказать, что в языке Ada программисты должны выбрать один стиль (либо полное вычисление, либо укороченное) для всей программы, используя другой стиль только в крайнем случае; в языке С вычисления всегда укороченные.

Укороченность вычисления существенна тогда, когда сама возможность вычислить отношение в составном выражении зависит от предыдущего отношения:

```
if (a /= 0) and then (b/a > 25) then .. .
```
Ada

Такая ситуация часто встречается при использовании указателей (гл. 8):

```
if (ptr /= null) and then (ptr.value = key) then . ..
```
Ada

6.3. Операторы цикла

Операторы цикла наиболее трудны для программирования: в них легко сделать ошибку, особенно на границах цикла, то есть при первом и последнем выполнении тела цикла. Кроме того, неэффективная программа чаще всего расходует большую часть времени в циклах, поэтому так важно понимать их реализацию. Структура цикла показана на рис. 6.3. Оператор цикла имеет точку *входа,* последовательность операторов, которые составляют цикл, и одну

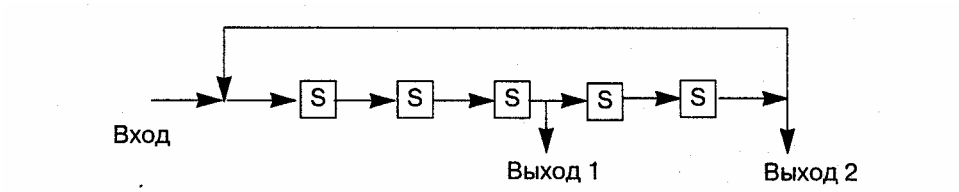

Рис. 6.3. Структура цикла.

или несколько точек *выхода.* Так как мы (обычно) хотим, чтобы наши циклы завершались, с точкой выхода бывает связано соответствующее *условие,* которое определяет, следует сделать выход или продолжить выполнение цикла. Циклы различаются числом, типом и расположением условий выхода. Мы начнем с обсуждения циклов с произвольными условиями выхода, называемыми циклами while, а в следующем разделе обсудим частный случай — циклы for.

Наиболее общий тип цикла имеет единственный выход в начале цикла, т.е. в точке входа. Он называется циклом while:

```
while (s[i]. data != key)
```
C

Цикл while прост и надежен. Поскольку условие проверяется в начале цикла, мы знаем, что тело цикла будет полностью выполнено столько раз, сколько потребуется по условию. Если условие выхода сначала имеет значение False,то тело цикла не будет выполнено, и это упрощает программирование граничных условий:

```
while (count > 0) process(s[count].data);
```
C

Если в массиве нет данных, выход из цикла произойдет немедленно.

Во многих случаях, однако, выход естественно писать в конце цикла. Так обычно делают, когда нужно инициализировать переменную перед каждым выполнением. В языке Pascal есть оператор повторения repeat:

```
repeat
    read(v);
     put_in_table(v);
until v = end_value;
```
Pascal

В языке Pascal repeat заканчивается, когда условие выхода принимает значение True. Не путайте его с циклом do в языке C, который заканчивается, когда условие выхода принимает значение False:

```
do{
    v = get();
    put_in_table(v);
   } while (v != end_value);
```
C

Принципы безупречного структурного программирования требуют, чтобы все выходы из цикла находились только в начале или конце цикла. Это делает программу более легкой для анализа и проверки. Но на практике бывают нужны выходы и из середины цикла, особенно при обнаружении ошибки:

```
while not found do
begin
(* Длинное вычисление *)
(* Обнаружена ошибка, выход *)
(* Длинное вычисление *)
end
```

Pascal

Pascal, в котором не предусмотрен выход из середины цикла, использует следующее неудовлетворительное решение: установить условие выхода и использовать if-оператор, чтобы пропустить оставшуюся часть цикла:

```
while not found do
begin
(* Длинное вычисление *)
   if error_detected then found := True
else
    begin
    (* Длинное вычисление *)
    end
end
```

Pascal

В языке C можно использовать оператор break:

```
while (!found) {
        /* Длинное вычисление */
        if (error_detected()) break;
         /* Длинное вычисление */
                  }
```

C

В Ada есть обычный цикл while, а также оператор exit, с помощью которого можно выйти из цикла в любом месте; как правило, пара связанных операторов if и exit заменяется удобной конструкцией when:

```
while not Found loop
   -- Длинное вычисление
exit when error_detected;
    - Длинное вычисление
 end loop;
```

Ada

Операционная система или система, работающая в реальном масштабе времени, по замыслу, не должна завершать свою работу, поэтому необходим способ задания бесконечных циклов. В Ada это непосредственно выражается оператором loop без условия выхода:

```
loop
…
end loop;
```

Ada

В других языках нужно написать обычный цикл с искусственным условием выхода, которое гарантирует, что цикл не завершится:

```
while(1==1){
…
}
```
C

Реализация

Цикл while:

```
while (expression)
statement;
```
C

реализуется так:

```
L1: compute      R1.expr
    jump_zero    R1,L2      Выйти из цикла, если false
    statement               Тело цикла
    jump         L1         Перейти на проверку завершения цикла L2:
```

Обратите внимание, что в реализации цикла while есть *две* команды перехода! Интересно, что если выход находится в конце цикла, то нужна только одна команда перехода:

```
do{
statement;
} while (expression);
```
C

компилируется в

```
L1:    statement
       compute     expr
       jump_nz     L1          Не ноль — это True
```

Хотя цикл while очень удобен с точки зрения читаемости программы, эффективность кода может быть увеличена путем замены его на цикл do. Для выхода из середины цикла требуются два перехода точно так же, как и для цикла while.

6.4. Цикл for

Очень часто мы знаем количество итераций цикла: это либо константа, известная при написании программы, либо значение, вычисляемое перед началом цикла. Цикл со счетчиком можно запрограммировать следующим образом:

```
int i;                          /* Индекс цикла */
int low, high;                  /* Границы цикла */

i = low;                        /* Инициализация индекса */
while (i <= high) {             /* Вычислить условие выхода */
      statement;
      i++:                      /* Увеличить индекс */
};
```
C

Поскольку это общая парадигма, постольку для упрощения программирования во всех (императивных) языках есть цикл for. Его синтаксис в языке С следующий:

```
int i;                          /* Индекс цикла */
 int low, high;                 /* Границы цикла */
```

C

```
for (i = low; i <= high; i++) {
    statement;
}
```

В Ada синтаксис аналогичный, за исключением того, что объявление и увеличение переменной цикла неявные:

```
Low, High: Integer;
```

Ada

```
for I in Low .. High loop
statement;
end loop;
```

Ниже в этом разделе мы обсудим причины этих различий.

Известно, что в циклах for легко сделать ошибки в значениях границ. Цикл выполняется для каждого из значений от low до high; таким образом, общее число итераций равно *high - low +1*. Однако, если значение low строго больше значения high, цикл будет выполнен ноль раз. Если вы хотите выполнить цикл точно Л/ раз, цикл for будет иметь вид:

Ada

```
forlinl ..N loop...
```

и число итераций равно *N* -1 + 1 = *N*. В языке С из-за того, что для массивов индексы должны начинаться с нуля, цикл со счетчиком обычно записывается так:

```
for(i = 0;i<n;i++)...
```

C

Так как запись i < *n* означает то же самое, что и i <= *(n - 1),* цикл выполняется *(n* -1)-0+1 =*n* раз, как и требуется.

Обобщения в циклах for

Несмотря на то, что все процедурные языки содержат циклы for, они значительно отличаются по предоставляемым дополнительным возможностям. Две крайности — это Ada и С.

В Ada исходным является положение, что цикл for должен использоваться *только с* фиксированным числом итераций и что это число можно вычислить перед началом цикла. Объясняется это следующим: 1) большинство реальных циклов простые, 2) другие конструкции легко запрограммировать в явном виде, и 3) циклы for и сами по себе достаточно трудны для тестирования и проверки. В языке Ada нет даже классического обобщения: увеличения переменной цикла на значения, отличные от 1 (или -1). Язык Algol позволяет написать итерации для последовательности нечетных чисел:

```
for I := 1 to N step 2 do ...
```

Algol

в то время как в Ada мы должны явно запрограммировать их вычисление:

```
for l in 1 .. (N + 1)/2 loop
```

Ada

```
|1 =2*|-1;
…
end loop;
```

В языке C все три элемента цикла for могут быть произвольными выражениями:

```
for(i=j*k;   (i<n)&&(j + k>m);    i + = 2*j)...
```

C

В описании C определено, что оператор

```
for (expression_1 ; expression_2; expression_3) statement;
```

C

эквивалентен конструкции

```
for(expression_1 ;
while (expression_2) {
          statement;
          expression_3;
}
```

C

В языке Ada также допускается использование выражений для задания границ цикла, но они вычисляются только один раз при входе в цикл. То есть

```
for I in expression_1 .. expression_2 loop
     statement;
end loop;
```

Ada

эквивалентно

```
I = expression_1;
Temp = expression_2;
while (I < Temp) loop
     statement;
      I: = I + 1;
end loop;
```

Ada

Если тело цикла изменяет значение переменных, используемых при вычислении выражения expression_2, то верхняя граница цикла в Ada изменяться не будет. Сравните это с данным выше описанием цикла for в языке C, который заново вычисляет значение выражения expression_2 на каждой итерации.

Обобщения в языке C — нечто большее, чем просто «синтаксический сахар», поскольку операторы внутри цикла, изменяющие выражения expres-sion_2 и expression_3, могут вызывать побочные эффекты. Побочных эффектов следует избегать по следующим причинам.

• Побочные эффекты затрудняют полную проверку и тестирование цикла.

• Побочные эффекты неблагоприятно воздействуют на читаемость и поддержку программы.

•Побочные эффекты делают цикл гораздо менее эффективным, потому что выражения expression_2 и expression_3 нужно заново вычислять на каждой итерации. Если побочных

эффектов нет, оптимизирующий компилятор может вынести эти вычисления за границу цикла.

Реализация

Циклы for — наиболее часто встречающиеся источники неэффективности в программах, потому что небольшие различия в языках или небольшие изменения в использовании оператора могут иметь серьезные последствия. Во многих случаях оптимизатор в состоянии решить эти проблемы, но лучше их понимать и избегать, чем доверяться оптимизатору. В этом разделе мы более подробно опишем реализацию на уровне регистров.

В языке Ada цикл

```
for I in expression_1 .. expression_2 loop
      statement;
end loop;
```

Ada

компилируется в

```
      compute        R1,expr_1
      store          R1,l          Нижняя граница индексации
      compute        R2,expr_2
      store          R2,High       Верхняя граница индексации
L1:   load           R1,l          Загрузить индекс
      load           R2,High       Загрузить верхнюю границу
      jump_gt        R1,R2,L2      Завершить цикл, если больше
      statement                    Тело цикла
      load           R1,l          Увеличить индекс
      incr           R1
      store          R1,l
      jump           L1
L2:
```

Очевидная оптимизация — это закрепление регистра за индексной переменной I и, если возможно, еще одного регистра за High:

```
      compute        R1 ,expr_1    Нижняя граница в регистре
      compute        R2,expr_2     Верхняя граница в регистре
L1:   jump_gt        R1,R2,L2      Завершить цикл, если больше
      statement
      incr R1                      Увеличить индексный регистр
      jump L1
 L2:
```

Рассмотрим теперь простой цикл в языке C:

```
for (i = expression_1 ; expression_2; i++)
 statement;
```

C

Это компилируется в

```
        compute     R1,expr_1
        store       R1,i            Нижняя граница индексации
L1:     compute     R2,expr_2       Верхняя граница внутри цикла!
        jump_gt     R1,R2,L2        Завершить цикл, если больше
        statement                   Тело цикла
        load        R1,i            Увеличить индекс
        incr        R1
        store       R1,i
        jump        L1
L2:
```

Обратите внимание, что выражение expression_2, которое может быть очень сложным, теперь вычисляется внутри цикла. Кроме того, выражение expres-sion_2 обязательно использует значение индексной переменной i, которая изменяется при каждой итерации. Таким образом, оптимизатор должен уметь выделить неизменяющуюся часть вычисления выражения expression_2, чтобы вынести ее из цикла.

Можно ли хранить индексную переменную только в регистре для увеличения эффективности? Ответ зависит от двух свойств цикла. В Ada индексная переменная считается константой и не может изменяться программистом. В языке C индексная переменная — это обычная переменная; она может храниться в регистре только в том случае, когда абсолютно *исключено* изменение ее текущего значения где-либо вне цикла. Никогда не используйте глобальную переменную в качестве индексной переменной, потому что другая процедура может прочитать или изменить ее значение:

C

```
int i;

void p2(void) {
   i = i + 5;
}
void p1(void) {
     for (i=0; i<100; i++)          /* Глобальная индексная переменная */
          p2();                     /* Побочный эффект изменит индекс*/
}
```

Второе свойство, от которого зависит оптимизация цикла, — потенциальная возможность использования индексной переменной за пределами цикла. В Ada индексная переменная неявно объявляется for-оператором и *недоступна* за пределами цикла. Таким образом, независимо от того, как осуществляется выход из цикла, мы не должны сохранять значение регистра. Рассмотрим следующий цикл поиска значения key в массиве a:

C

```
inta[100];
int i, key;

key = get_key();
for(i = 0;i< 100; i++)
     if (a[i] == key) break;
process(i);
```

Переменная i должна содержать правильное значение независимо от способа, которым был сделан выход из цикла. Это может вызывать затруднения при попытке оптимизировать код. Обратите внимание, что в Ada требуется явное кодирование для достижения того же самого результата, потому что индексная переменная не существует вне области цикла:

Ada

```
Found: Integer := False;

for I in 1 ..100 loop
    if A(l) = Key then
       Found = I;
       exit;
    end if;
end loop;
```

Определение области действия индексов цикла в языке C++ с годами менялось, но конечное определение такое же, как в Ada: индекс не существует вне области цикла:

C++

```
for(int i=0;i<100;i++){
            // Индексная переменная является локальной для цикла
}
```

На самом деле в любом управляемом условием операторе (включая, if- и switch-операторы) можно задать в условии несколько объявлений; область их действия будет ограничена управляющим оператором. Это свойство может способствовать читаемости и надежности программы, предотвращая непреднамеренное использование временного имени.

6.5. «Часовые»

Следующий раздел не касается языков программирования как таковых; скорее, он предназначен для того, чтобы показать, что программу можно улучшить за счет более совершенных алгоритмов и методов программирования, не прибегая к «игре» на языковых частностях. Этот раздел включен в книгу, потому что тема выхода из цикла при последовательном переборе является предметом интенсивных дебатов, однако существует и другой алгоритм, который является одновременно ясным, надежным и эффективным.

В последнем примере предыдущего раздела (поиск в массиве) есть три команды перехода в каждой итерации цикла: условный переход цикла for, условный переход if-оператора и переход от конца цикла обратно к началу. Проблема поиска в данном случае состоит в том, что мы проверяем сразу два условия: найдено ли значение key и достигнут ли конец массива? Используя *«часового» (sentinel)* *, мы можем два условия заменить одним. Идея состоит в том, чтобы ввести в начале массива дополнительно еще один элемент («часового») и хранить в нем эталонное значение key, которое нужно найти в массиве (рис. 6.4).

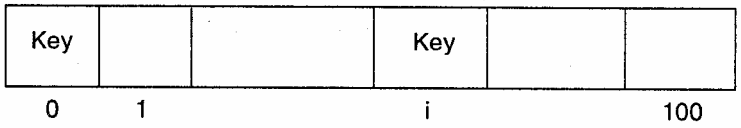

Рис. 6.4. «Часовые».

Поскольку мы обязательно найдем key либо как элемент массива, либо как искусственно введенный элемент, постольку достаточно проверять только одно условие внутри цикла:

Ada

```
type A_Type is array(0 .. 100) of Integer;
                        -- Дополнительное место в нулевой позиции для «часового»
function Find_Key(A: A_Type; Key: Integer)
        return Integer is
      I:  Integer := 100;                    -- Поиск с конца
begin
    A(0) := Key;                             -- Установить «часового»
    while A(l) /= Key loop
    I:=I-1;
    end loop;
     return I;
end Find_Key;
```

Если при возврате управления из функции значение I равно нулю, то Key в массиве нет; в противном случае I содержит индекс найденного значения. Этот код более эффективен, цикл чрезвычайно прост и может быть легко проверен.

6.6. Инварианты

Формальное определение семантики операторов цикла базируется на концепции *инварианта:* формулы, которая остается истинной после каждого выполнения тела цикла. Рассмотрим предельно упрощенную программу для вычисления целочисленного деления a на b с тем, чтобы получить результат y:

C

```
y = 0;
x = a;
while (x >- b) {                          /* Пока b «входит» в x, */
x -= b;                                   /* вычитание b означает, что */
y++;                                      /* результат должен быть увеличен */
}
```

и рассмотрим формулу:

$$a = yb + x$$

где курсивом обозначено значение соответствующей программной переменной. После операторов инициализации она, конечно, будет правильной, поскольку y = 0 и $x = a$. Кроме того, в конце программы формула *определяет*, что y есть результат целочисленного деления a/b при условии, что остаток x меньше делителя b.

Не столь очевидно то, что формула остается правильной после каждого выполнения тела цикла. В такой тривиальной программе этот факт легко увидеть с помощью простой арифметики, изменив значения x и y в теле цикла:

$$(y + \backslash)b + (x\text{-}b) = yb + b + x - b = yb + x = a$$

Таким образом, выполнение тела цикла переводит программу из состояния, которое удовлетворяет инварианту, в другое состояние, которое по-прежнему удовлетворяет инварианту.

Теперь заметим: для того чтобы завершить цикл, булево условие в цикле while должно иметь значение False, то есть вычисление должно быть в таком состоянии, при котором *--(x > b)*, что эквивалентно *x < b*. Объединив эту формулу с инвариантом, мы показали, что программа действительно выполняет целочисленное деление.

Точнее, *если* программа завершается, *то* результат является правильным. Это называется *частичной правильностью*. Чтобы доказать *полную правильность*, мы должны также показать, что цикл завершается.

Это делается следующим образом. Так как во время выполнения программы b является константой (и предполагается положительной!), нам нужно показать, что неоднократное уменьшение x на b должно, в конечном счете, привести к состоянию, в котором *0 < x < b*. Но 1) поскольку *x* уменьшается неоднократно, его значение не может бесконечно оставаться больше значения b; 2) из условия завершения цикла и из вычисления в теле цикла следует, что x никогда не станет отрицательным. Эти два факта доказывают, что цикл должен завершиться.

Инварианты цикла в языке Eiffel

Язык Eiffel имеет в себе средства для задания контрольных утверждений вообще (см. раздел 11.5) и инвариантов циклов в частности:

```
from
        y = 0; x = a;
invariant
        a = yb + x
variant
         x
until
          x< b
 loop
          x :=x-b;
          y:=y+1;
 end
```

Eiffel

Конструкция from устанавливает начальные условия, конструкция until задает условие для завершения цикла, а операторы между loop и end образуют тело цикла. Конструкция invariant определяет инвариант цикла, а конструкция variant определяет выражение, которое будет уменьшаться (но останется неотрицательным) с каждой итерацией цикла. Правильность инварианта проверяется после каждого выполнения тела цикла.

6.7. Операторы goto

В первоначальном описании языка Fortran был только один структурированный управляющий оператор: оператор do, аналогичный циклу for. Все остальные передачи управления делались с помощью условных или безусловных переходов на метки, т. е. с помощью операторов, которые называются goto:

```
                if(a.eq.b)goto12
                …
                goto 5
4               …                                          Fortan
                …
12              …
                …
5               …
                if (x .gt. y) goto 4
```

В 1968 г. Э. Дейкстра написал знаменитое письмо, озаглавленное «оператор goto следует считать вредным», с которого началась дискуссия о структурном программировании. Основной аргумент против goto состоит в том, чтс произвольные переходы не структуированы и создают «программу-спагетти», в которой возможные пути выполнения так переплетаются, что ее невозможно понять и протестировать. Аргументом в пользу goto является то что в реальных программах часто требуются более общие управляющие структуры, чем те, которые предлагают структурированные операторы, и чтс принуждение программистов использовать их приводит к искусственному и сложному коду.

Оглядываясь назад, можно сказать, что эти дебаты были чересчур эмоциональны и затянуты, потому что основные принципы совсем просты и не требуют долгого обсуждения. Более того, в современные диалекты языка Fortrar добавлены более совершенные операторы управления с тем, чтобы оператор goto больше не доминировал.

Можно доказать математически, что достаточно if- и while-операторов чтобы записать любую необходимую управляющую структуру. Кроме того эти операторы легко понять и использовать. Различные синтаксические расширения типа циклов for вполне ясны и при правильном использовании не представляют никаких трудностей для понимания или сопровождения программы. Так почему же языки программирования (включая Ada, при разработке которого исходили из соображений надежности) сохраняют goto?

Причина в том, что есть несколько вполне определенных ситуаций, где лучше использовать goto. Во-первых, многие циклы не могут завершаться в точке входа, как того требует цикл while. Попытка превратить все циклы в циклы while может затемнить суть дела. В современных языках гибкости привносимой операторами exit и break, достаточно и оператор goto для этой цели обычно не нужен. Однако goto все еще существует и иногда может быть полезным. Обратите внимание, что как язык C, так и Ada, ограничивают применение goto требованием, чтобы метка находилась в той же самой процедуре.

Вторая ситуация, которую легко запрограммировать с помощью goto, — выход из глубоко вложенного вычисления. Предположим, что глубоко внутри ряда вызовов процедур обнаружена ошибка, что делает неверным все вычисление. В этой ситуации естественно было бы запрограммировать выдачу сообщения об ошибке и завершить или возвратить в исходное состояние все вычисление. Однако для этого требуется сделать возврат из многих процедур, которые должны знать, что произошла ошибка. Проще и понятнее выйти в основную программу с помощью goto.

В языке C нет никаких средств для обработки этой ситуации (не подходит даже goto по причине ограниченности рамками отдельной процедуры), поэтому для обработки серьезных ошибок нужно использовать средства операционной системы. В языках Ada, C++ и Eiffel есть специальные языковые конструкции, так называемые *исключения (exseptions)*, см. гл. 11, которые непосредственно решают эту проблему. Таким образом, операторы goto в большинстве случаев были вытеснены по мере совершенствования языков.

Назначаемые goto-операторы

В языке Fortran есть конструкция, которая называется *назначаемым (assigned) оператором* goto. Можно определить метку-переменную и присваивать ей значение той или иной конкретной метки. При переходе по метке-переменной фактической целевой точкой перехода является значение этой переменной:

Fortan

```
      assign 5 to Label
         ...
      if (x .gt. y) assign 6 to Label
5        ...
6        ...
       goto Label
```

Проблема, конечно, в том, что присваивание значения метке-переменной могло быть сделано за миллионы команд до того, как выполняется goto, и программы в таких случаях отлаживать и верифицировать практически невозможно.

Хотя в других языках не существует назначаемого goto, такую конструкцию совсем просто смоделировать, определяя много небольших подпрограмм и манипулируя передачами указателей на эти подпрограммы. Соотносить конкретный вызов с присвоением указателю значения, связывающего его с той или иной подпрограммой, достаточно трудно. Поэтому указатели на подпрограммы следует применять лишь в случаях хорошей структурированности, например в таблицах указателей на функции в интерпретаторах или в механизмах обратного вызова.

6.8. Упражнения

1. Реализует ваш компилятор все case-/switch-операторы одинаково, или он пытается выбирать оптимальную реализацию для каждого оператора?

2. Смоделируйте оператор repeat языка Pascal в Ada и C.

3. Первоначально в языке Fortran было определено, что цикл выполняется, по крайней мере, один раз, даже если значение low больше, чем значение high! Чем могло быть мотивировано такое решение?

4. Последовательный поиск в языке C:

C

```
while (s[i].data != key)
      i++;
```

можно записать как

C

```
while (s[i++].data != key)
;                                   /* Пустой оператор */
```

В чем различие между двумя вариантами вычислений?

5. Предположим, что в языке Ada переменная индекса может существовать за рамками цикла. Покажите, как бы это воздействовало на оптимизацию цикла.

6. Сравните сгенерированный для поиска код, реализованный с помощью операторов break или exit, с кодом, сгенерированным для поиска с «часовым».

7. Напишите программу поиска с «часовым», используя do-while вместо while. Будет ли это эффективнее?

8. Почему мы помещали «часового» в начало массива, а не в конец?

9. (Шолтен) В игре Го используют камни двух цветов, черные и белые. Предположим, что у вас в коробке неизвестная смесь камней, и вы выполняете следующий алгоритм:

Ada

```
while Stones_Left_in_Can loop              -- пока есть камни в коробке
    Remove_Two_Stones(S1, S2);             -- вынуть два камня
    if Color(S1 )=Color(S2) then
      Add_Black_Stone;                     --добавить черный камень
    else
      Add_White_Stone;                     -- добавить белый камень
    end if;
end loop;
```

Найдите переменную, значение которой уменьшается, оставаясь неотрицательным, и тем самым покажите, что цикл заканчивается. Можете ли вы что-нибудь сказать относительно цвета последнего камня? (Подсказка: напишите инвариант цикла для числа белых камней).

Глава 7

Подпрограммы

7.1. Подпрограммы: процедуры и функции

Подпрограмма — это сегмент программы, к которому можно обратиться из любого места внутри программы. Подпрограммы используются по разным причинам:

• Сегмент программы, который должен выполняться на разных стадиях вычисления, может быть написан один раз в виде подпрограммы, а затем многократно выполняться. Это экономит память и позволяет избежать ошибок, возможных при копировании кода с одного места на другое.

• Подпрограмма — это логическая единица декомпозиции программы. Даже если сегмент выполняется только один раз, полезно оформить его в виде подпрограммы с целью тестирования, документирования и улучшения читаемости программы.

• Подпрограмму также можно использовать как физическую единицу декомпозиции программы, т. е. как единицу компиляции. В языке Fortran подпрограмма (subroutine) — это единственная единица и декомпозиции, и компиляции. В современных языках физической единицей декомпозиции является модуль, представляющий собой группу объявлений и подпрограмм (см. гл. 13).

Подпрограмма состоит из:

• объявления, которое задает интерфейс с подпрограммой; это объявление включает имя подпрограммы, список параметров (если есть) и тип возвращаемого значения (если есть);

• локальных объявлений, которые действуют только внутри тела подпрограммы;

• последовательности выполняемых операторов.

Локальные объявления и выполняемые операторы образуют *тело* подпрограммы. Подпрограммы, которые возвращают значение, называются *функциями (functions), а* те, что не возвращают, — *процедурами (procedures).* Язык С не имеет отдельного синтаксиса для процедур; вместо этого следует написать функцию, которая возвращает тип void, т.е. тип без значения:

C

```
void proc(int a, float b);
```

Такая функция имеет те же свойства, что и процедура в других языках, поэтому мы используем термин «процедура» и при обсуждении языка С.

Обращение к процедуре задается оператором *вызова* процедуры call. В языке Fortran он имеет специальный синтаксис:

C

```
call proc(x,y)
```

тогда как в других языках просто пишется имя процедуры с фактическими параметрами:

C

```
proc(x,y);
```

Семантика вызова процедуры следующая: приостанавливается текущая последовательность команд; выполняется последовательность команд внутри тела процедуры; после завершения тела процедуры выполнение продолжается с первой команды, следующей за вызовом процедуры. Это описание игнорирует передачу параметров и их области действия, что будет объектом детального рассмотрения в следующих разделах.

Так как функция возвращает значение, объявление функции должно определять тип возвращаемого значения. В языке C тип функции задается в объявлении функции перед ее именем:

C

```
int func(int a, float b);
```

тогда как в языке Ada используется другой синтаксис:

Ada

```
function Func(A: Integer; B: Float) return Integer;
```

Вызов функции является не оператором, а элементом *выражения:*

C

```
a = x + func(r,s) + y;
```

Тип результата функции не должен противоречить типу, ожидаемому в выражении. Обратите внимание, что в языке C во многих случаях делаются неявные преобразования типов, тогда как в Ada тип результата должен точно соответствовать контексту. По смыслу вызов функции аналогичен вызову процедуры: приостанавливается вычисление выражения; выполняются команды тела функции; затем возвращенное значение используется для продолжения вычисления выражения.

Термин «функция» фактически совершенно не соответствует тому контексту, в котором он употребляется в обычных языках программирования. В математике функция — всего лишь отображение одного набора значений на другой. Если использовать техническую терминологию, то математическая функция *не имеет побочного эффекта*, потому что ее «вычисление» прозрачно в точке, в которой делается «вызов». Если есть значение 3.6, и вы запрашиваете значение sin(3.6), то вы будете получать один и тот же результат всякий раз, когда в уравнении встретится эта функция. В программировании функция может выполнять произвольное вычисление, включая ввод-вывод или изменение глобальных структур данных:

C

```
int x,y,z;
intfunc(void)
{
        y = get();                  /* Изменяет глобальную переменную */
        return x*y;                 /* Значение зависит от глобальной переменной */
z = x + func(void) + y;
```

Если оптимизатор изменил порядок вычисления так, что x + y вычисляется перед вызовом функции, то получится другой результат, потому что функция изменяет значение y.

Поскольку все подпрограммы в C — функции, в программировании на языке C широко используются возвращаемые значения и в «невычислительных» случаях, например в подпрограммах ввода-вывода. Это допустимо при условии, что понятны возможные трудности, связанные с зависимостью от порядка и оптимизацией. Исследование языков программирования привело к разработке интереснейших языков, которые основаны на математически правильном понятии функции (см. гл. 16).

7.2. Параметры

В предыдущем разделе мы определили подпрограммы как сегменты кода, которые можно неоднократно вызывать. Практически всегда при вызове требуется выполнять код тела подпрограммы для новых данных. Способ повлиять на выполнение тела подпрограммы состоит в том, чтобы «передать» ей необходимые данные. Данные передаются подпрограмме в виде последовательности значений, называемых *параметрами.* Это понятие взято из математики, где для функции задается последовательность *аргументов:* sin (2piK). Есть два понятия, которые следует четко различать:

• *Формальный параметр* — это объявление, которое находится в объявлении подпрограммы. Вычисление в теле подпрограммы пишется в.терми-нах формальных параметров.

• *Фактический параметр* — это значение, которое вызывающая программа передает подпрограмме.

В следующем примере:

```
int i,,j;
char a;
void p(int a, char b)
{
      i = a + (int) b;
}
```

C

```
P(i,a);
P(i+j, 'x');
```

формальными параметрами подпрограммы p являются a и b, в то время как фактические параметры при первом вызове — это i и a, а при втором вызове — i + j и 'x'.

На этом примере можно отметить несколько важных моментов. Во-первых, так как фактические параметры являются значениями, то они могут быть константами или выражениями, а не только переменными. Даже когда переменная используется как параметр, на самом деле подразумевается «текущее значение, хранящееся в переменной». Во-вторых, *пространство имен* у разных подпрограмм разное. Тот факт, что первый формальный параметр называется a, не имеет отношения к остальной части программы, и этот параметр может быть переименован, при условии, конечно, что будут переименованы все вхождения формального параметра в теле подпрограммы. Переменная a, объявленная вне подпрограммы, полностью независима от переменной с таким же именем, объявленной внутри подпрограммы. В разделе 7.7 мы более подробно рассмотрим связь между переменными, объявленными в разных подпрограммах.

Установление соответствия параметров

Обычно фактические параметры при вызове подпрограммы только перечисляются, а соответствие их формальным параметрам определяется по позиции параметра:

```
procedure Proc(First: Integer; Second: Character);
 Proc(24, 'X');
```
Ada

Однако в языке Ada при вызове возможно использовать установление соответствия по имени, когда каждому фактическому параметру предшествует имя формального параметра. Следовать порядку объявления параметров при этом не обязательно:

```
Ada Proc(Second => 'X', First => 24);
```
Ada

Обычно этот вариант используется вместе с параметрами по умолчанию, причем параметры, которые не написаны явно, получают значения по умолчанию, заданные в объявлении подпрограммы:

```
procedure Proc(First: Integer := 0; Second: Character := '*');
Proc(Second => 'X');
```
Ada

Соответствие по имени и параметры по умолчанию обычно используются в командных языках операционных систем, где каждая команда может иметь множество параметров и обычно необходимо явно изменить только некоторые из них. Однако этот стиль программирования таит в себе ряд опасностей. Использование параметров по умолчанию может сделать программу трудной для чтения, потому что синтаксически отличающиеся обращения фактически вызывают одну и ту же подпрограмму. Соответствие по имени является проблематичным, потому что при этом зависимость объявления подпрограммы и вызовов оказывается более сильной, чем это обычно требуется. Если при вызовах библиотечных подпрограмм вы пользуетесь только позиционными параметрами, то вы могли бы купить библиотеку у конкурирующей фирмы и просто перекомпилировать или перекомпоновать программу:

```
X:=Proc_1 (Y) + Proc_2(Z);
```
Ada

Однако если вы используете именованные параметры, то, возможно, вам придется сильно изменить свою программу, чтобы установить соответствие новым именам параметров:

```
X := Proc_1(Parm => Y) + Proc_2(Parm => Z);
```
Ada

7.3. Передача параметров подпрограмме

Описание механизма передачи параметров — один из наиболее тонких и важных аспектов спецификации языка программирования. Неверная передача параметров — главный источник серьезных ошибок, поэтому мы подробно рассмотрим этот вопрос.

Давайте начнем с данного выше определения: значение фактического параметра передается формальному параметру. Формальный параметр — это просто переменная, которая объявлена внутри подпрограммы, поэтому, очевидно, нужно копировать значение фактического параметра в то место памяти, которое выделено для формального параметра. Этот механизм называется

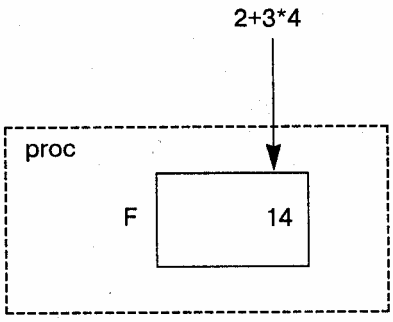

Рис. 7.1. Семантика copy-in.

«семантикой copy-in» («копирование в») или *«вызовом по значению» (call-by-value).* На рисунке 7.1 показана семантика copy-in для процедуры:

```
procedure Proc(F: in Integer) is
begin
    ... ;
end Proc;
```

Ada

и вызова:

Ada

```
Proc(2+3*4);
```

Преимущества семантики copy-in:

• Copy-in является самым надежным механизмом передачи параметров. Поскольку передается только копия фактического параметра, подпрограмма никак не может испортить фактический параметр, который, несомненно, «принадлежит» вызывающей программе. Если подпрограмма изменяет формальный параметр, изменяется только копия, а не оригинал.

• Фактические параметры могут быть константами, переменными или выражениями.

• Механизм copy-in может быть очень эффективным, потому что начальные затраты на копирование делаются один раз, а все остальные обращения к формальному параметру на самом деле являются обращениями к локальной копии. Как мы увидим в разделе 7.7, обращение к локальным переменным чрезвычайно эффективно.

Если семантика copy-in настолько хороша, то почему существуют другие механизмы? дело в том, что часто мы хотим изменить фактический параметр, несмотря на тот факт, что такое изменение «небезопасно»:

• Функция возвращает только один результат, но, если результат вычисления достаточно сложен, может возникнуть желание вернуть несколько значений. Чтобы сделать это, необходимо задать в процедуре несколько фактических параметров, которым могут быть присвоены результаты вычисления. Обратите внимание, что этого часто можно избежать, определив функцию, которая возвращает в качестве результата запись.

• Кроме того, цель выполнения подпрограммы может состоять в модификации данных, которые ей передаются, а не в их вычислении. Обычно это происходит, когда подпрограмма обрабатывает структуру данных. Например, подпрограмма, сортирующая массив, не вычисляет значение; ее цель состоит только в том, чтобы изменить фактический параметр. Нет никакого смысла сортировать копию массива!

• Параметр может быть настолько большим, что копировать его неэффективно. Если copy-in используется для массива из 50000 целых чисел, может просто не хватить памяти, чтобы сделать копию, или затраты на копирование будут чересчур большими.

Первые две ситуации легко разрешить с помощью *семантики copy-out* («копирование из»). Фактический параметр должен быть переменной, а подпрограмме передается адрес фактического параметра, который она сохраняет. Для формального параметра используется временная локальная переменная, и значение должно быть присвоено формальному параметру, по крайней мере, один раз во время выполнения подпрограммы. Когда выполнение подпрограммы завершено, значение копируется в переменную, на которою указывает сохраненный адрес. На рисунке 7.2 показана семантика copy-out для следующей подпрограммы:

```
procedure Proc(F: out Integer) is
begin
   F := 2+3*4;                          -- Присвоение параметру
end Proc;

A: Integer;
Proc(A);                                -- Вызов процедуры с переменной
```

Ada

Когда нужно модифицировать фактический параметр, как, например, в sort, можно использовать *семантику copy-in/out* фактический параметр копирует-

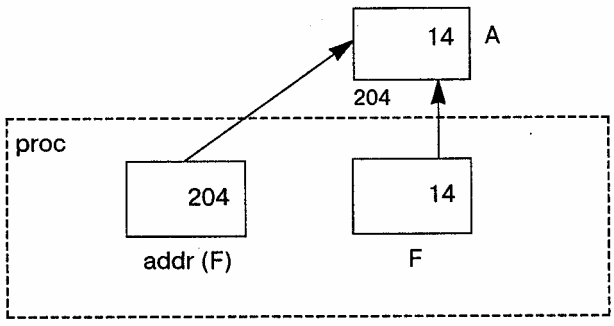

Рис. 7.2. Семантика copy-out.

ся в подпрограмму, когда она вызывается, а результирующее значение копируется обратно после ее завершения.

Однако механизмы передачи параметров на основе копирования не могут решить проблему эффективности, связанную с «большими» параметрами. Решение, которое известно как *«вызов по ссылке»* (call-by-reference) или *«семантика ссылки»* (reference cemantics), состоит в том, чтобы передать адрес фактического параметра и обращаться к параметру косвенно (см. рис. 7.3). Вызов подпрограммы эффективен, потому что для каждого параметра передается только указатель небольшого, фиксированного размера; однако обращение к параметру может оказаться неэффективным из-за косвенности.

Чтобы получить доступ к фактическому параметру, нужно загрузить его адрес, а затем выполнить дополнительную команду для загрузки значения. Обратите внимание, что при использовании семантики ссылки (или copy-out), фактический параметр должен быть переменной, а не выражением, так как ему будет присвоено значение.

Другая проблема, связанная с вызовом по ссылке, состоит в том, что может возникнуть *совмещение имен (aliasing),* т. е. может возникнуть ситуация, в которой одна и та же переменная известна под несколькими именами.

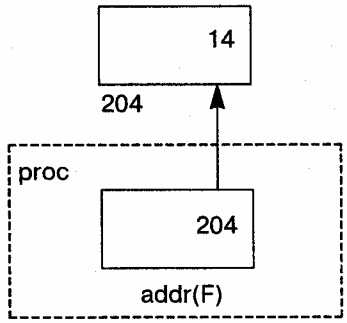

Рис. 7.3. Семантики ссылки.

В следующем примере внутри функции f переменная global получает *алиас* (т. е. альтернативное имя) *parm:

```
int global = 4;
 inta[10];                                                    C

int f(int *parm)
 {
        *parm = 5:                          /* Та же переменная, что и "global" */
          return 6;
}

x = a[global] + f(&global);
```

В этом примере, если выражение вычисляется в том порядке, в котором оно записано, его значение равно a[4] + 6, но из-за совмещения имен значение выражения может быть 6 + a[5], если компилятор при вычислении выражения выберет порядок, при котором вызов функции предшествует индексации массива. Совмещение имен часто приводит к непереносимости программ.

Реальный недостаток «вызова по ссылке» состоит в том, что этот механизм по сути своей ненадежен. Предположим, что по некоторым причинам подпрограмма считает, что фактический параметр — массив, тогда как реально это всего лишь одно целое число. Это может привести к тому, что будет затерта некоторая произвольная область памяти, так как подпрограмма работает с фактическим параметром, а не просто с локальной копией. Этот тип ошибки встречается очень часто, потому что подпрограмма обычно пишется не тем программистом, который разрабатывает вызывающую программу, и всегда возможно некоторое недопонимание.

Безопасность передачи параметров можно повысить с помощью строгого контроля соответствия типов, который гарантирует, что типы формальных и фактических параметров совместимы. Однако все еще остается возможность недопонимания между тем программистом, кто написал подпрограмму, и тем, чьи данные модифицируются. Таким образом, мы имеем превосходный механизм передачи параметров, который не всегда достаточно эффективен (семантика copy-in), а также необходимые, но ненадежные механизмы (семантика copy-out и семантика ссылки). Выбор усложняется ограничениями, которые накладывают на программиста различные языки программирования. Теперь мы подробно опишем механизмы передачи параметров для нескольких языков.

Параметры в языках C и C++

В языке C есть только один механизм передачи параметров — copy-in:

```
int i = 4;                                          /* Глобальная переменная */
```

C

```
void proc(int i, float f)
{
i=i+(int) f;                                        /* Локальная переменная "i" */
}
proc(j, 45.0);                                      /* Вызов функции */
```

В proc изменяемая переменная i является локальной копией, а не глобальной переменной i. Чтобы получить функциональные возможности семантики ссылки или copy-out, пишущий на C программист должен прибегать к явному использованию указателей:

```
int i = 4; /* Глобальная переменная */                    [c]
void proc(int *i, float f)
{
*i = *i+ (int) f; /* Косвенный доступ */
}
proc(&i, 45.0); /* Понадобилась операция получения адреса */
```

После выполнения proc значение глобальной переменной i изменится. Необходимость пользоваться указателями для реализации ссылочной семантики следует отнести к неудачным решениям в языке C, потому что начинающим программистам приходится изучать это относительно сложное понятие в начале курса.

В языке C++ этот недостаток устранен, поскольку в нем есть возможность задавать параметры специального ссылочного типа *(reference parameters):*

```
int i = 4;                                  // Глобальная переменная
```

C++

```
void proc(int & i, float f)
{
     i = i + (int) f;                       // Доступ по ссылке
}

proc(i, 45.0);                              // Не нужна операция получения адреса
```

Обратите внимание на естественность стиля программирования, при котором нет неестественного использования указателей. Это усовершенствование механизма передачи параметров настолько важно, что оправдывает использование C++ в качестве замены C.

Вам часто придется применять указатели в C или ссылки в C++ для передачи больших структур данных. Конечно, в отличие от копирования параметров (copy-in), существует опасность случайного изменения фактического параметра. Можно задать для параметра доступ только для чтения, объявив его константой:

```
void proc(const Car_Data & d)
{
    d.fuel = 25;                            // Ошибка, нельзя изменять константу
}
```

Объявления const следует использовать возможно шире, как для того, чтобы сделать смысл параметров более прозрачным для читателей программы, так и для того, чтобы отлавливать возможные ошибки.

Другая проблема, связанная с параметрами в языке С, состоит в том, что массивы не могут быть параметрами. Если нужно передать массив, передается адрес первого элемента массива, а процедура отвечает за правильный доступ к массиву. Для удобства имя массива в качестве параметра автоматически рассматривается как указатель на первый элемент:

```
intb[50];                          /* Переменная типа массив */
void proc(int a[ ])                /* "Параметр-массив" */
{                                                                  C
a[100] = a[200];                   /* Сколько элементов? */
 }
proc(&b[0]);                       /* Адрес первого элемента */
proc(b);                           /* Адрес первого элемента */
```

Программисты, пишущие на С, быстро к этому привыкают, но, все равно, это является источником недоразумений и ошибок. Проблема состоит в том, что, поскольку параметр — это, фактически, указатель на отдельный элемент, то допустим *любой* указатель на переменную того же типа:

```
int i;
void proc(int a[ ]);               /* "Параметр-массив" */
proc(&i);                          /* Допустим любой указатель на целое число!! */
```

Наконец, в языке С контроль соответствия типов никак не действует между файлами, поэтому можно в одном файле поместить

```
[C] void proc(float f) { ...}      /* Описание процедуры */        C
```

а в другом файле —

```
void proc(int i);                  /* Объявление процедуры */
proc(100);                                                         C
```

а затем месяцами искать ошибку.

Язык С++ требует выполнения контроля соответствия типов для параметров. Однако он не требует, чтобы реализации включали библиотечные средства, как в Ada (см. раздел 13.3), которые могут гарантировать контроль соответствия типов для независимо компилируемых файлов. Компиляторы С++ выполняют контроль соответствия типов вместе с компоновщиком: типы параметров шифруются во внешнем имени подпрограммы (процесс называется *name mangling),* а компоновщик следит за тем, чтобы связывание вызовов с программами делалось только в случае корректной сигнатуры параметров. К сожалению, этот метод не может охватывать все возможные случаи несоответствия типов.

Параметры в языке Pascal

В языке Pascal параметры передаются по значению, если явно не задана передача по ссылке:

```
procedure proc(P_lnput: Integer; var P_0utput: Integer);
```
Pascal

Ключевое слово var указывает, что параметр вызывается по ссылке, в противном случае используется вызов по значению, даже если параметр очень большой. Параметры могут быть любого типа, включая массивы, записи или другие сложные структуры данных. Единственное ограничение состоит в том, что тип результата функции должен быть скалярным. Типы фактических параметров проверяются на соответствие типам формальных параметров.

Как мы обсуждали в разделе 5.3, в языке Pascal есть серьезная проблема, связанная с тем, что границы массива рассматриваются как часть типа. Для решенения этой проблемы стандарт Pascal определяет *совместимые параметры массива* (conformant array parameters).

Параметры в языке Ada

В языке Ada принят новый подход к передаче параметров. Она определяется в терминах предполагаемого использования, а не в терминах механизма реализации. Для каждого параметра нужно явно выбрать один из трех возможных *режимов*.

in — Параметр можно читать, но не писать
(значение по умолчанию).
out — Параметр можно писать, но не читать.

in out — Параметр можно как читать, так и писать.

Например:

```
procedure Put_Key(Key: in Key_Type);
procedure Get_Key(Key: out Key_Type);
procedure Sort_Keys(Keys: in out Key_Array);
```
Ada

В первой процедуре параметр Key должен читаться с тем, чтобы его можно было «отправить» (Put) в структуру данных (или на устройство вывода). Во второй значение получено (Get) из структуры данных, а после завершения процедуры значение присваивается параметру. Массив Keys, который нужно отсортировать, должен быть передан как in out, потому что сортировка включает и чтение, и запись данных массива.

Для функций в языке Ada разрешена передача параметров только в режиме in. Это не делает функции Ada функциями без побочных эффектов, потому что нет никаких ограничений на доступ к глобальным переменным; но это может помочь оптимизатору увеличить эффективность вычисления выражения.

Несмотря на то что режимы определены не в терминах механизмов реализации, язык Ада определяет некоторые требования по реализации. Параметры элементарного типа (числа, перечисления и указатели) должны передаваться соответствующим копированием: copy-in для in-параметров, copy-out для out-параметров и copy-in/out для in-out-параметров. Реализация режимов для составных параметров (массивов и записей) не определена, и компилятор может выбрать любой механизм. Это приводит к тому, что правильность программы в Ada может зависеть от выбранного механизма реализации, поэтому такие программы непереносимы.

Между формальными и фактическими параметрами делается строгий контроль соответствия типов. Тип фактического параметра должен быть таким же, как и у формального; неявное преобразование типов никогда не выполняется. Однако, как мы

обсуждали в разделе 5.3, подтипы не обязаны быть идентичными, пока они совместимы; это позволяет передавать произвольный массив формальному неограниченному параметру.

Параметры в языке Fortran

Мы вкратце коснемся передачи параметров в языке Fortran, потому что здесь возможны эффектные ошибки. Fortran может передавать только скалярные значения; интерпретация формального параметра, как массива, выполняется вызванной подпрограммой. Для всех параметров используется передача параметра по ссылке. Более того, каждая подпрограмма компилируется независимо, и не делается никакой проверки на совместимость между объявлением подпрограммы и ее вызовом.

В языке определено, что если делается присваивание формальному параметру, то фактический параметр должен быть переменной, но из-за независимой компиляции это правило не может быть проверено компилятором. Рассмотрим следующий пример:

```
Subroutine Sub(X, Y)
Real X,Y
X=Y
End
Call Sub(-1.0,4.6)
```

Fortran

У подпрограммы два параметра типа Real. Поскольку используется семантика ссылки, Sub получает указатели на два фактических параметра, и присваивание выполняется непосредственно для фактических параметров (см. рис. 7.4). В результате область памяти, где хранится значение -1,0, изменяется! Без преувеличения можно сказать, что выявить и устранить эту ошибку буквально

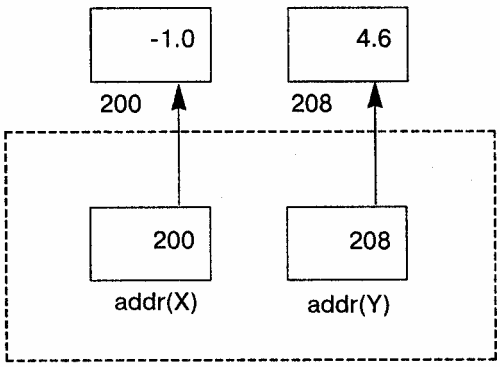

Рис. 7.4. Изменение константы в Fortran.

нет никаких средств, так как отладчики позволяют проверять и отслеживать только переменные, но не константы. Как показывает практика, правильное соответствие фактических и формальных параметров — краеугольный камень надежного программирования.

7.4. Блочная структура

Блок — это объект, состоящий из объявлений и выполняемых операторов. Аналогичное определение было дано для тела подпрограммы, и точнее будет сказать, что тело подпрограммы — это блок. Блоки вообще и процедуры в частности могут быть вложены один в другой. В этом разделе будут обсуждаться взаимосвязи вложенных блоков.

Блочная структура была сначала определена в языке Algol, который включает как процедуры, так и неименованные блоки. В языке Pascal есть вложенные процедуры, но нет неименованных блоков; в C есть неименованные блоки, но нет вложенных процедур; а Ada поддерживает и то, и другое.

Неименованные блоки полезны для ограничения области действия переменных, так как позволяют объявлять их, когда необходимо, а не только в начале подпрограммы. В свете современной тенденции уменьшения размера подпрограмм полезность неименованных блоков падает.

Вложенные процедуры можно использовать для группировки операторов, которые выполняются в нескольких местах внутри подпрограммы, но обращаются к локальным переменным и поэтому не могут быть внешними по отношению к подпрограмме. До того как были введены модули и объектно-ориентированное программирование, для структурирования больших программ использовались вложенные процедуры, но это запутывает программу и поэтому не рекомендуется.

Ниже приведен пример полной Ada-программы:

```
procedure Mam is
   Global: Integer;

   procedure Proc(Parm: in Integer) is
      Local: Integer;
   begin
      Global := Local + Parm;
   end Proc;

begin -- Main
  Global := 5;
   Proc(7);
   Proc(8);
end Main;
```

Ada

Ada-программа — это *библиотечная* процедура, то есть процедура, которая не включена внутрь никакого другого объекта и, следовательно, может храниться в Ada-библиотеке. Процедура начинается с объявления процедуры Main, которое служит описанием интерфейса процедуры, в данном случае внешним именем программы. Внутри библиотечной процедуры есть два объявления: переменной Global и процедуры Proc. После объявлений располагается последовательность исполняемых операторов главной процедуры. Другими словами, процедура Main состоит из объявления процедуры и блока. Точно так же локальная процедура Proc состоит из объявления процедуры (имени процедуры и параметров) и блока, содержащего объявления переменных и исполняемые операторы. Говорят, что Proc — процедура *локальная* для Main или *вложенная* внутри Main.

С каждым объявлением связаны три свойства.

Область действия. Область действия переменной — это сегмент программы, в котором она определена.

Видимость. Переменная видима внутри некоторого подсегмента области действия, если к ней можно непосредственно обращаться по имени.

Время жизни. Время жизни переменной — это период выполнения программы, в течение которого переменной выделена память.

Обратите внимание, что время жизни — динамическая характеристика поведения программы при выполнении, в то время как область действия и видимость касаются исключительно статического текста программы.

Продемонстрируем эти абстрактные определения на приведенном выше примере. Область действия переменной начинается в точке объявления и заканчивается в конце блока, в котором она определена. Область действия переменной Global включает всю программу, тогда как область действия переменной Local ограничена отдельной процедурой. Формальный параметр Parm рассматривается как локальная переменная, и его область действия также ограничена процедурой.

Видимость каждой переменной в этом примере идентична ее области действия; к каждой переменной можно непосредственно обращаться во всей ее области действия. Поскольку область действия и видимость переменной Local ограничены локальной процедурой, следующая запись недопустима:

```
begin — Main
Global := Local + 5;                    -- Local здесь вне области действия
end Main;
```

Ada

Однако область действия переменной Global включает локальную процедуру, поэтому обращение внутри процедуры корректно:

```
procedure Proc(Parm: in Integer) is
   Local: Integer;
begin
    Global := Local + Parm;             --Global здесь в области действия
end Proc;
```

Время жизни переменной — от начала выполнения ее блока до конца выполнения этого блока. Блок процедуры Main — вся программа, поэтому переменная Global существует на протяжении выполнения программы. Такая переменная называется *статической:* после того как ей отведена память, она существует до конца программы. Локальная переменная имеет два времени жизни, соответствующие двум вызовам локальной процедуры. Так как эти интервалы не перекрываются, переменной каждый раз можно выделять новое место памяти. Локальные переменные называются автоматическими, потому что память для них *автоматически* выделяется при вызове процедуры (при входе в блок) и освобождается при возврате из процедуры (при выходе из блока).

Скрытые имена

Предположим, что имя переменной, которое используется в главной программе, повторяется в объявлении в локальной процедуре:

```
procedure Mam is
      Global: Integer;
      V: Integer;                              -- Объявление в Main

procedure Proc(Parm: in Integer) is
      Local: Integer;
      V: Integer;                              -- Объявление в Proc
begin
    Global := Local + Parm + V;                -- Какое именно V используется?
end Proc;

begin -- Main
    Global := Global + V;                      -- Какое именно V используется?
end Main;
```

В этом случае говорят, что локальное объявление *скрывает (или перекрывает)* глобальное объявление. Внутри процедуры любая ссылка на V является ссылкой на локально объявленную переменную. С технической точки зрения область действия глобальной переменной V простирается от точки объявления до конца Main, но она невидима в локальной процедуре Proc.

Скрытие имен переменных внутренними объявлениями удобно тем, что программист может многократно использовать естественные имена типа Current_Key и не должен изобретать странно звучащие имена. Кроме того, всегда можно добавить глобальную переменную, не беспокоясь о том, что ее имя совпадет с именем какой-нибудь локальной переменной, которое используется одним из программистов вашей группы. Недостаток же состоит в том, что имя переменной могло быть случайно перекрыто, особенно если используются большие включаемые файлы для централизации глобальных объявлений, поэтому, вероятно, лучше избегать перекрытия имен переменных. Однако нет никаких возражений против многократного использования имени в разных областях действия, так как нельзя получить доступ к обеим переменным одновременно независимо от того, являются имена одинаковыми или разными:

```
procedure Main is
  procedure Proc_1 is
          Index: Integer;                      -- Одна область действия
          …
  endProc_1;
    procedure Proc_2 is
       Index: Integer;                         -- Неперекрывающаяся область действия
         …
       end Proc_2;
begin – Main
…
end Main;
```

Ada

Глубина вложения

Принципиальных ограничений на глубину вложения нет, но ее может произвольно ограничивать компилятор. Область действия и видимость определяются правилами, данными выше: область действия переменной — от точки ее объявления до конца блока, а видимость — такая же, если только не скрыта внутренним объявлением. Например:

Ada

```
procedure Main is
    Global: Integer;
    procedure Level_1 is
     Local: Integer;                    -- Внешнее объявление Local
procedure Level_2 is

        Local: Integer;                 --Внутреннее объявление Local
      begin -- Level_2
        Local := Global;                -- Внутренняя Local скрывает внешнюю Local
      end Level_2;

begin -- Level_1
    Local := Global;                    -- Только внешняя Local в области действия
    Level_2:
end Level_1;

begin -- Main
   Level_1;
   Level_2;                             -- Ошибка, процедура вне области действия
end Main;
```

Область действия переменной Local, определенной в процедуре Level_1, простирается до конца процедуры, но она скрыта внутри процедуры Level_2 объявлением того же самого имени.

Считается, что сами объявления процедуры имеют область действия и видимость, подобную объявлениям переменных. Таким образом, область действия Level_2 распространяется от ее объявления в Level_1 до конца Level_1. Это означает, что Level_1 может *вызывать* Level_2, даже если она не может обращаться к переменным внутри Level_2. С другой стороны, Main не может непосредственно вызывать Level_2, так как она не может обращаться к объявлениям, которые являются локальными для Level_1.

Обратите внимание на возможность запутаться из-за того, что обращение к переменной Local в теле процедуры Level_1 отстоит от объявления этой переменной *дальше* по тексту программы, чем объявление Local, заключенной внутри процедуры Level_2. В случае многочисленных локальных процедур найти правильное объявление бывает трудно. Чтобы избежать путаницы, лучше всего ограничить глубину вложения двумя или тремя уровнями от уровня главной программы.

Преимущества и недостатки блочной структуры

Преимущество блочной структуры в том, что она обеспечивает простой и эффективный метод декомпозиции процедуры. Если вы избегаете чрезмерной вложенности и скрытых переменных, блочную структуру можно использовать для написания надежных программ, так как связанные локальные процедуры можно хранить и обслуживать вместе. Особенно важно блочное структурирование при выполнении сложных вычислений:

```
procedure Proc(...) is
            -- Большое количество объявлений
begin
   -- Длинное вычисление 1
if N < 0 then
            -- Длинное вычисление 2, вариант 1
elsif N = 0 then
            -- Длинное вычисление 2, вариант 2
else
            -- Длинное вычисление 2, вариант 3
end if;
     -- Длинное вычисление 3
end Proc;
```

Ada

В этом примере мы хотели бы не записывать три раза Длинное вычисление 2, а оформить его как дополнительную процедуру с одним параметром:

```
procedure Proc(...) is
         -- Большое количество объявлений
        procedure Long_2(l: in Integer) is
        begin
              -- Здесь действуют объявления Proc
        end Long_2;
begin
    -- Длинное вычисление 1
if N<0thenl_ong_2(1);
elsif N = 0 then Long_2(2);
else Long_2(3);
end if;
    -- Длинное вычисление 3
end Proc;
```

Ada

Однако было бы чрезвычайно трудно сделать Long_2 независимой процедурой, потому что пришлось бы передавать десятки параметров, чтобы она могла обращаться к локальным переменным. Если Long_2 — вложенная процедура, то нужен только один параметр, а к другим объявлениям можно непосредственно обращаться в соответствии с обычными правилами для области действия и видимости.

Недостатки блочной структуры становятся очевидными, когда вы пытаетесь запрограммировать большую систему на таком языке, как стандарт Pascal, в котором нет других средств декомпозиции программы.

• Небольшие процедуры получают чрезмерную «поддержку». Предположим, что процедура, преобразующая десятичные цифры в шестнадцате-ричные, используется во многих глубоко вложенных процедурах. Такаясервисная процедура должна быть определена в некотором общем предшествующем элементе. На практике в больших программах с блочной структурой проявляется тенденция появления большого числа небольших сервисных процедур, описанных на самом высоком уровне объявлений. Это делает текст программы неудобным для работы, потому что нужную программу бывает просто трудно разыскать.

• Защита данных скомпрометирована. Любая процедура, даже та, объявление которой в структуре глубоко вложено, может иметь доступ к глобальным переменным. В большой программе, разрабатываемой группой программистов, это приводит к тому, что ошибки, сделанные младшим членом группы, могут привести к скрытым ошибкам. Эту ситуацию можно сравнить с компанией, где каждый служащий может свободно обследовать сейф в офисе начальника, а начальник не имеет права проверять картотеки младших служащих!

Эти проблемы настолько серьезны, что каждая коммерческая реализация Pascal определяет (нестандартную) структуру модуля, чтобы иметь возможность создавать большие проекты. В главе 13 мы подробно обсудим конструкции, которые применяются для декомпозиции программы в таких современных языках, как Ada и C++. Однако блочная структура остается важным инструментом детализированного программирования отдельных модулей.

Понимать блочную структуру важно также и потому, что языки программирования реализованы с использованием стековой архитектуры, которая непосредственно поддерживает блочную структуру (см. раздел 7.6).

7.5. Рекурсия

Чаще всего (процедурное) программирование использует *итерации*, то есть циклы; однако *рекурсия* — описание объекта или вычисления в терминах самого себя — является более простым математическим понятием, а также мощной, но мало используемой техникой программирования. Здесь мы рассмотрим, как программировать рекурсивные подпрограммы.

Наиболее простой пример рекурсии — функция, вычисляющая факториал. Математически она определяется как:

$$0! = 1$$
$$n! = n \times (n - 1)!$$

Это определение сразу же переводится в программу, которая использует рекурсивную функцию:

C

```
int factorial(int n)
{
if (n == 0) return 1 ;
else return n * factorial(n - 1);
}
```

Какие свойства необходимы для поддержки рекурсии?

• Компилятор должен выдавать *чистый код*. Так как при каждом обращении к функции factorial используется одна и та же последовательность машинных команд, код не должен изменять сам себя.

• Должна существовать возможность выделять во время выполнения произвольное число ячеек памяти для параметров и локальных переменных.

Первое требование выполняется всеми современными компиляторами. Самоизменяющийся код — наследие более старых стилей программирования и используется редко. Обратите внимание, что если программа предназначена для размещения в постоянной памяти (ROM), то она не может изменяться по определению.

Второе требование определяется временем жизни локальных переменных. В примере время жизни формального параметра n — с момента, когда процедура вызвана, до ее завершения. Но до завершения процедуры делается еще один вызов, и этот вызов требует, чтобы была выделена память для нового формального параметра. Чтобы вычислять factorial(4), выделяется память для 4, затем 3 и т. д., всего пять ячеек. Нельзя выделить память перед выполнением, потому что ее количество зависит от параметра функции во время выполнения. В разделе 7.6 показано, как это требование выделения памяти непосредственно поддерживается стековой архитектурой.

Большинство программистов обратит внимание, что функцию, вычисляющую факториал, можно написать так же легко и намного эффективнее с помощью итерации:

```
int factorial(int n)
{
     int i = n;
     result = 1;                                                    C
     while (i != 0) {
          result = result * i;
               i--;
}
          return result;
}
```

Так почему же используют рекурсию? Дело в том, что многие алгоритмы можно изящно и надежно написать с помощью рекурсии, в то время как итерационное решение трудно запрограммировать и легко сделать ошибки. Примером служат алгоритм быстрой сортировки и алгоритмы обработки древовидных структур данных. Языковые понятия, рассматриваемые в гл. 16 и 17 (функциональное и логическое программирование), опираются исключительно на рекурсию, а не на итерацию. Даже для обычных языков типа C и Ada рекурсию, вероятно, следует использовать более часто, чем это делается, из-за краткости и ясности программ, которые получаются в результате.

7.6. Стековая архитектура

Стек — это структура данных, которая принимает и выдает данные в порядке LIFO — Last-In, First-Out (последним пришел, первым вышел). Конструкции LIFO существуют в реальном мире, например стопка тарелок в кафетерии или пачка газет в магазине. Стек может быть реализован с помощью массива или списка (см. рис. 7.5). Преимущество списка в том, что он не имеет границ, а его размер ограничен только общим объемом доступной памяти. Массивы же намного эффективнее и неявно используются при реализации языков программирования.

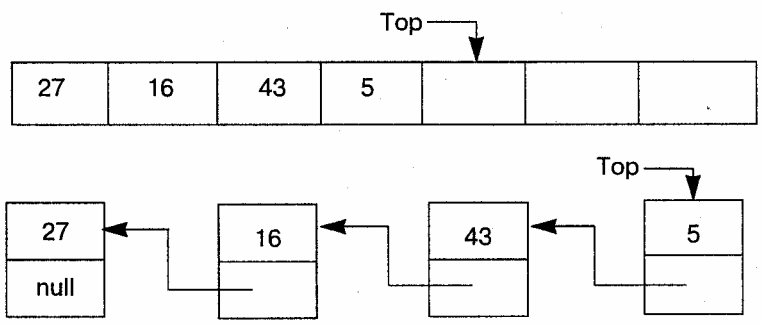

Рис.7.5. Реализации стека.

Кроме массива (или списка) в состав стека входит еще один элемент — *указатель вершины стека* (top-of-stack pointer). Это индекс первой доступной пустой позиции в стеке. Вначале переменная top будет указывать на первую позицию в стеке. На стеке допустимы две операции — push (поместить в стек) и pop (извлечь из стека), push — это процедура, получающая элемент как параметр, который она помещает в вершину стека, увеличивая указатель вершины стека top. pop — это функция, которая возвращает верхний элемент стека, уменьшая top, чтобы указать, что эта позиция стала новой пустой позицией.

Следующая программа на языке С реализует стек целых чисел, используя массив:

```
#define Stack_Size 100
int stack[Stack_Size];
int top = 0;
void push(int element)
{
        if (top == Stack_Size)          /* Переполнение стека, предпримите
                                         что-нибудь! * I
else stack[top++] = element;
 }
int pop(void)
{
if (top == 0)                           /* Выход за нижнюю границу стека,
                                         предпримите то-нибудь! */
else return stack[--top];
 }
```

C

Выход за нижнюю границу стека произойдет, когда мы попытаемся извлечь элемент из пустого стека, а переполнение стека возникнет при попытке поместить элемент в полный стек. Выход за нижнюю границу стека всегда вызывается ошибкой программирования, поскольку вы сохраняете что-нибудь в стеке тогда и только тогда, когда предполагаете извлечь это позднее. Переполнение может происходить даже в правильной программе, если объем памяти недостаточен для вычисления.

Выделение памяти в стеке

Как используется стек при реализации языка программирования? Стек нужен для хранения информации, касающейся вызова процедуры, включая локальные переменные и параметры, для которых память автоматически выделяется после входа в процедуру и освобождается после выхода. Стек является подходящей структурой данных потому, что входы и выходы в процедуры делаются в порядке LIFO, а все нужные данные принадлежат процедуре, которая в цепочке вызовов встречается раньше.
Рассмотрим программу с локальными процедурами:

```
procedure Main is
    G: Integer;
     procedure Proc_1 is
          L1: Integer;
     begin ... end Proc_1 ;

     procedure Proc_2 is
          L2: Integer;
      begin... end Proc_2;
begin
    Proc_1;
    Proc_2;
end Main;
```

Ada

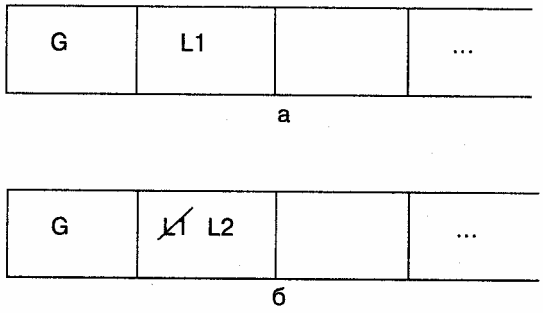

Рис. 7.6. Распределение памяти в стеке.

Когда начинает выполняться Main, должна быть выделена память для G. Когда вызывается Proc_1, должна быть выделена дополнительная память для L1 без освобождения памяти для G (см. рис. 7.6а). Память для L1 освобождается перед выделением памяти для L2, так как Proc_1 завершается до вызова Proc_2 (см. рис. 7.66). Вообще, независимо оттого, каким образом процедуры вызывают друг друга, первый элемент памяти, который освобождается, является последним занятым элементом, поэтому память для переменных и параметров может отводиться в стеке.

Рассмотрим теперь вложенные процедуры:

```
procedure Main is
   G: Integer;

    procedure Proc_1 (P1: Integer) is
       L1: Integer;

    procedure Proc_2(P2: Integer) is
        L2: Integer;
    begin
       L2 := L1 + G + P2;
    end Proc_2;
  begin -- Proc_1
       Proc_2(P1);
   end Proc_1;
begin -- Main
      Proc_1 (G);
end Main;
```

Ada

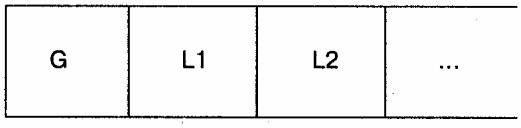

Рис. 7.7. Вложенные процедуры.

Proc_2 может вызываться только из Proc_1. Это означает, что Proc_1 еще не завершилась, ее память не освобождена, и место, выделенное для L1, должно все еще оставаться занятым (см. рис. 7.7). Конечно, Proc_2 завершается раньше Proc_1, которая в свою очередь завершается раньше Main, поэтому память может быть освобождена с помощью операции pop.

Записи активации

Фактически стек используется для поддержки всего вызова процедуры, а не только для размещения локальных переменных. Сегмент стека, связанный с каждой процедурой, называется *записью активации (activation record)* для процедуры. Вкратце, вызов процедуры реализуется следующим образом (см. рис. 7.8):

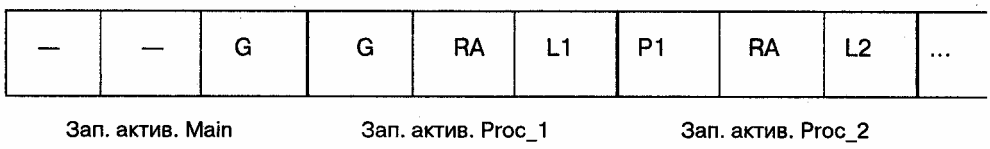

Рис. 7.8. Записи активации.

1. В стек помещаются фактические параметры. К ним можно обращаться по смещению от начала записи активации.

2. В стек помещается *адрес возврата RA (return address).* Адрес возврата — это адрес оператора, следующего за вызовом процедуры.

3. Индекс вершины стека увеличивается на общий объем памяти, требуемой для хранения локальных переменных.

4. Выполняется переход к коду процедуры.

После завершения процедуры перечисленные шаги выполняются в обратном порядке:

1. Индекс вершины стека уменьшается на величину объема памяти, выделенной для локальных переменных.

2. Адрес возврата извлекается из стека и используется для восстановления указателя команд.

3. Индекс вершины стека уменьшается на величину объема памяти, выделенной для фактических параметров.

Хотя этот алгоритм может показаться сложным, на большинстве компьютеров он фактически может быть выполнен очень эффективно. Объем памяти для переменных, параметров и дополнительной информации, необходимой для организации вызова процедуры, известен на этапе компиляции, а описанная технология всего лишь требует изменения индекса стека на константу,

Доступ к значениям в стеке

В классическом стеке единственно допустимые операции — это push и pop. «Рабочий» стек, который мы описали, — более сложная структура, потому что мы хотим иметь эффективный доступ не только к самому последнему значению, помещенному в стек, но и ко всем локальным переменным и ко всем параметрам. В частности, необходимо иметь возможность обращаться к этим данным относительно индекса вершины стека:

stack[top -25]; C

Однако стек может содержать и другие данные помимо тех, что связаны с вызовом процедуры (например, временные переменные, см. раздел 4.7), поэтому обычно

поддерживается еще дополнительный индекс, так называемый *указатель дна (bottom pointer),* который указывает на начало записи активации (см. раздел 7.7). Даже если индекс вершины стека изменится во время выполнения процедуры, ко всем данным в записи активации можно обращаться по фиксированным смещениям от указателя дна стека.

Параметры

Существуют два способа реализации передачи параметров. Более простым является помещение в стек самих параметров (либо значений, либо ссылок). Этот способ используется в языках Pascal и Ada, потому что в этих языках номер и тип каждого параметра известны на этапе компиляции. По этой информации смещение каждого параметра относительно начала записи активации может быть вычислено во время компиляции, и к каждому параметру можно обращаться по этому фиксированному смещению от указателя дна стека:

```
load    R1 ,bottom_pointer        Указатель дна
add     R1 ,#offset-of-parameter   + смещение
load    R2,(R1)                   Загрузить значение, адрес которого
                                   находится в R1
```

Если указатель дна сохраняется в регистре, то этот код обычно можно сократить до одной команды. При выходе из подпрограммы выполняется *очистка стека* подпрограммой, которая сбрасывает указатель вершины стека так, чтобы параметры фактически больше не находились в стеке.

При использовании этого метода в языке C возникает проблема, связанная с тем, что C разрешает иметь в процедуре переменное число параметров:

```
void proc(int num_args,...);
```
C

Так как количество параметров подпрограмме неизвестно, она не может очистить стек. Ответственность за очистку стека, таким образом, перекладывается на вызыватель, который знает, сколько параметров было передано. Это приводит к некоторому перерасходу памяти, потому что код очистки стека теперь свой при *каждом* вызове вместо того, чтобы быть общим для всех вызовов.

Когда число параметров неизвестно, возможен альтернативный способ передачи параметров, при котором фактические параметры сохраняются в отдельном блоке памяти, а затем передается адрес этого блока в стек. Для доступа к параметру требуется дополнительная косвенная адресация, поэтому этот метод менее эффективен, чем непосредственное помещение параметров в стек.

Обратите внимание, что иногда нельзя сохранить параметр непосредственно в стеке. Как вы помните, формальный параметр в языке Ada может иметь неограниченный тип массива, границы которого неизвестны во время компиляции:

```
procedure Proc(S: in String);
```
Ada

Таким образом, фактический параметр не может быть помещен непосредственно в стек. Вместо него в стек помещается дескриптор массива (dope vector) (см. рис. 5.4), который содержит указатель на массив.

Рекурсия

Архитектура стека непосредственно поддерживает рекурсию, поскольку каждый вызов процедуры автоматически размещает новую копию локальных переменных и параметров. Например, при каждом рекурсивном вызове функции факториала требуется одно слово памяти для параметра и одно слово памяти для адреса возврата. То, что издержки на рекурсию больше, чем на итерацию, связано с дополнительными командами, затрачиваемыми на вход в процедуру и выход из нее. Некоторые компиляторы пытаются выполнить оптимизацию, называемую оптимизацией хвостовой рекурсии *(tail-recursion)* или оптимизацией *последнего вызова (last-call).* Если единственный рекурсивный вызов в процедуре — последний оператор процедуры, то можно автоматически перевести рекурсию в итерацию.

Размер стека

Если рекурсивных вызовов нет, то теоретически перед выполнением можно просчитать общий размер используемого стека, суммируя размеры записей активации для каждой возможной цепочки вызовов процедур. Даже в сложной программе вряд ли будет трудно сделать приемлемую оценку этого числа. Добавьте несколько тысяч слов про запас, и вы вычислите размер стека, который наверняка не будет переполняться.

Однако при применении рекурсии размер стека во время выполнения теоретически неограничен:

C

```
i = get();
j = factorial(i);
```

В упражнениях приведена функция Акерманна, которая гарантированно переполнит любой стек! Но на практике обычно нетрудно оценить размер стека, даже когда используется рекурсия. Предположим, что размер записи активации приблизительно равен 10, а глубина рекурсии не больше нескольких сотен. Добавления к стеку лишних 10 Кбайт более чем достаточно.

Читатели, которые изучали структуры данных, знают, что рекурсией удобно пользоваться при работе с древовидными структурами в таких алгоритмах, как быстрая сортировка и приоритетные очереди. Глубина рекурсии в алгоритмах обработки древовидных структур данных — приблизительно $\log_2$ от размера структуры. Для реальных программ глубина рекурсии не превышает 10 или 20, поэтому опасность переполнения стека очень невелика.

Независимо от того, используется рекурсия или нет, сама природа рассматриваемых систем такова, что потенциально возможное переполнение стека должно как-то обрабатываться. Программа может либо полностью игнорировать эту возможность и при критических обстоятельствах разрушаться, либо проверять размер стека перед каждым вызовом процедуры, что может быть накладно. Компромиссное решение состоит в том, чтобы периодически проверять размер стека и предпринимать некоторые действия, если он стал меньше некоторого предельного значения, скажем 1000 слов.

7.7. Еще о стековой архитектуре

Доступ к переменным на промежуточных уровнях

Мы обсудили, как можно эффективно обращаться к локальным переменным по фиксированным смещениям от указателя дна, указывающего на запись активации. К глобальным данным, т. е. данным, объявленным в главной программе, также можно обращаться эффективно. Это легко увидеть, если рассматривать глобальные данные как локальные для главной процедуры. Память для глобальных данных распределяется при входе в главную процедуру, т. е. в начале программы. Так как их размещение известно на этапе компиляции, точнее, при компоновке, то действительный адрес каждого элемента известен или непосредственно, или как смещение от фиксированной позиции. На практике глобальные данные обычно распределяются независимо (см. раздел 8.3), но в любом случае адреса фиксированы.

Труднее обратиться к переменным на промежуточных уровнях вложения.

```
procedure Main is
    G: Integer,

    procedure Proc_1 is
      L1: Integer;

        procedure Proc_2 is                                    Ada
            L2: Integer;
        begin L2 := L1 + G; end Proc_2;
          procedure Proc_3 is
              L3: Integer;
          begin L3 := L1 + G; Proc_2; end Proc_3;

      begin -- Proc_1
          Proc_3;
      end Proc_1;

  begin — Main
    Proc_1;
  end Main;
```

Мы видели, что доступ к локальной переменной L3 и глобальной переменной G является простым и эффективным, но как можно обращаться к L1 в Proc_3? Ответ прост: значение указателя дна сохраняется при входе в процедуру и используется как указатель на запись активации объемлющей процедуры Proc_1. Указатель дна хранится в известном месте и может быть немедленно загружен, поэтому дополнительные затраты потребуются только на косвенную адресацию.

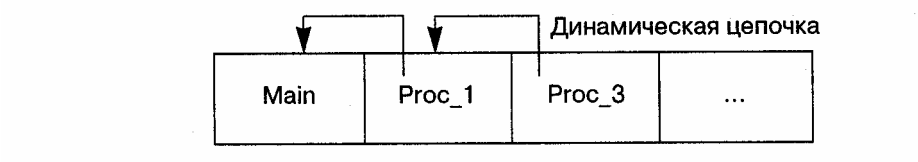

Рис. 7.9. Динамическая цепочка.

При более глубоком вложении каждая запись активации содержит указатель на предыдущую запись активации. Эти указатели на записи активации образуют *динамическую цепочку* (см. рис. 7.9). Чтобы обратиться к *вышележащей переменной* (вложенной менее глубоко), необходимо «подняться» по динамической цепочке. Связанные с этим затраты снижают эффективность работы с переменными промежуточных уровней при большой глубине вложенности. Обращение непосредственно к предыдущему уровню требует только одной косвенной адресации, и эпизодическое глубокое обращение тоже не должно вызывать никаких проблем, но в циклы не следует включать операторы, которые далеко возвращаются по цепочке.

Вызов вышележащих процедур

Доступ к промежуточным переменным фактически еще сложнее, потому что процедуре разрешено вызывать другие процедуры, которые имеют такой же или более низкий уровень вложения. В приведенном примере Proc_3 вызывает Proc_2. В записи активации для Proc_2 хранится указатель дна для процедуры Proc_3 так, что его можно восстановить, но переменные Proc_3 *недоступны* в Proc_2 в соответствии с правилами области действия.

Так или иначе, программа должна быть в состоянии идентифицировать *статическую цепочку,* т.е. цепочку записей активации, которая определяет статический контекст процедур согласно правилам области действия, в противоположность динамической цепочке вызовов процедур во время выполнения. В качестве крайнего случая рассмотрим рекурсивную процедуру: в динамической цепочке могут быть десятки записей активации (по одной для каждого рекурсивного вызова), но статическая цепочка будет состоять только из текущей записи и записи для главной процедуры.

Одно из решений состоит в том, чтобы хранить в записи активации статический уровень вложенности каждой процедуры, потому что компилятор знает, какой уровень необходим для каждого доступа. Если главная программа в примере имеет нулевой уровень, то обе процедуры Proc_2 и Proc_3 находятся на уровне 2. При продвижении вверх по динамической цепочке уровень вложенности должен уменьшаться на единицу, чтобы его можно было рассматривать как часть статической цепочки; таким образом, запись для Proc_3 пропускается, и следующая запись, уже запись для Proc_1 на уровне 1, используется, чтобы получить индекс дна.

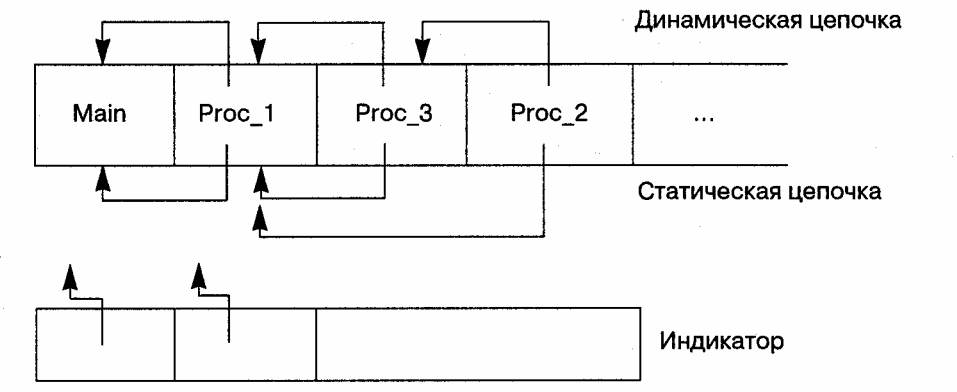

Рис. 7.10. Переменные на промежуточных уровнях.

Другое решение состоит в том, чтобы явно включить статическую цепочку в стек. На рисунке 7.10 показана статическая цепочка сразу после вызова Proc_2 из Proc_3 . Перед вызовом статическая цепочка точно такая же, как динамическая, а после вызова она стала короче динамической и содержит только главную процедуру и Proc_1.

Преимущество явной статической цепочки в том, что она часто короче, чем динамическая (вспомните здесь о предельном случае рекурсивной процедуры). Однако мы все еще должны осуществлять поиск при каждом обращении к промежуточной переменной. Более эффективное решение состоит в том, чтобы использовать *индикатор*, который содержит текущую статическую цепочку в виде массива, индексируемого по уровню вложенности (см. рис. 7.10). В этом случае для обращения к переменной промежуточного уровня, чтобы получить указатель на правильную запись активации, в качестве индекса используется уровень вложенности, затем из записи извлекается указатель дна, и, наконец, прибавляется смещение, чтобы получить адрес переменной. Недостаток индикатора в том, что необходимы дополнительные затраты на его обновление при входе и выходе из процедуры.

Возможная неэффективность доступа к промежуточным переменным не должна служить препятствием к применению вложенных процедур, но программистам следует учитывать такие факторы, как глубина вложенности, и правильно находить компромисс между использованием параметров и прямым доступом к переменным.

7.8. Реализация на процессоре Intel 8086

Чтобы дать более конкретное представление о реализации идей стековой архитектуры, рассмотрим вход в процедуру и выход из нее на уровне машинных команд для процессора серии Intel 8086. В качестве примера возьмем:

```
procedure Main is
      Global: Integer;
       procedure Proc(Parm: in Integer) is
             Local'1, Local2: Integer;
       begin
             Local2 := Global + Farm + Local 1 ;
       end Proc;
begin
      Proc(15);
end Main;
```

Ada

Процессор 8086 имеет встроенные команды push и pop, в которых подразумевается, что стек растет от старших адресов к младшим. Для стековых операций выделены два регистра: регистр sp, который указывает на «верхний» элемент в стеке, и регистр bp, который является указателем дна и идентифицирует местоположение начала записи активации.

При вызове процедуры в стек помещается параметр и выполняется команда вызова (call):

```
mov     ax, #15     Загрузить значение параметра
push    ax          Сохранить параметр в стеке
call    Proc        Вызвать процедуру
```

На рисунке 7.11 показан стек после выполнения этих команд — параметр и адрес возврата помещены в стек.

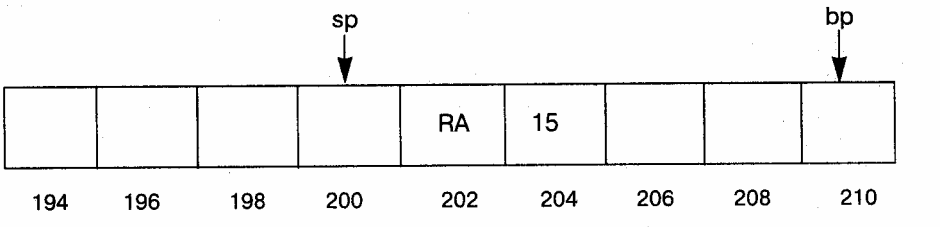

Рис. 7.11. Стек перед входом в процедуру.

Следующие команды являются частью кода процедуры и выполняются при входе в процедуру; они сохраняют старый указатель дна (динамическая связь), устанавливают новый указатель дна и выделяют память для локальных переменных, уменьшая указатель стека:

```
push    bp          Сохранить старый динамический указатель
mov     bp, sp      Установить новый динамический указатель
sub     sp,#4       Выделить место для локальных переменных
```

Получившийся в результате стек показан на рис. 7.12.

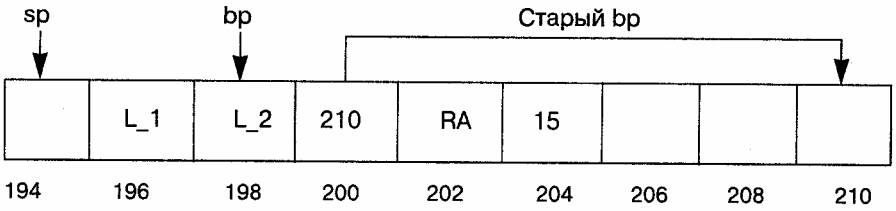

Рис. 7.12. Стек после входа в процедуру.

Теперь можно выполнить тело процедуры:

```
mov     ax,ds:[38]      Загрузить переменную Global
add     ax,[bp+06]      Прибавить параметр Parm
add     ax,[bp-02]      Прибавить переменную Local 1
mov     ax,[bp]         Сохранить в переменной Local2
```

Обращение к глобальным переменным делается через смещения относительно специальной области памяти, на которою указывает регистр ds (сегмент данных). К параметру Parm, который располагается в стеке «ниже» начала записи активации, обращаются при *положительном* смещении относительно bp. К локальным переменным, которые в стеке располагаются «выше», обращаются при *отрицательном* смещении относительно bp. Важно обратить внимание, что поскольку процессор 8086 имеет регистры и способы адресации, разработанные для обычных вычислений с использованием стека, то ко всем этим переменным можно обращаться одной командой.

При выходе из процедуры должны быть ликвидированы все изменения, сделанные при входе в процедуру:

```
mov     sp,bp       Очистить все локальные переменные
pop     bp          Восстановить старый динамический указатель
ret     2           Вернуться и освободить память параметров
```

Указатель вершины стека принимает значение указателя дна и таким образом действительно освобождает память, выделенную для локальных переменных. Затем старый динамический указатель выталкивается (pop) из стека, и bp теперь указывает на предыдущую запись активации. Остается только выйти из процедуры, используя адрес возврата, и освободить память, выделенную для параметров. Команда ret выполняет обе эти задачи; операнд команды указывает, сколько байтов памяти, выделенных для параметра, необходимо вытолкнуть из стека.

Подведем итог: как для входа, так и для выхода из процедуры требуется только по три коротких команды, и доступ к локальным и глобальным переменным и к параметрам является эффективным.

7.9. Упражнения

1. Использует ли ваш компилятор Ada значения или ссылки для передачи массивов и записей?

2. Покажите, как реализуется оптимизация последнего вызова процедуры при рекурсиях. Можно ли выполнить эту оптимизацию для функции факториала?

3. Функция Маккарти определяется следующей рекурсивной функцией:

```
function M(l: Integer) return Integer is
begin
        if I > 100 then return 1-10;
        else return M(M(I + 11));
end M;
```

Ada

а) Напишите программу для функции Маккарти и вычислите M(l) для 80</<110.

б) Смоделируйте вручную вычисление для M(91), показав рост стека.

в) Напишите итерационную программу для функции Маккарти.

4. Функция Акерманна определяется следующей рекурсивной функцией:

```
function A(M, N: Natural) return Natural is
begin
    if M = 0 then return N + 1 ;
    elsif N = 0 then return A(M -1,1);
    else return A(M - 1, A(M, N-1));
end A;
```

Ada

а) Напишите программу для функции Акерманна и проверьте, что A(0,0)=1, A(1,1)=3, A(2,2)=7, A(3,3)=61.

б) Смоделируйте вручную вычисление для A(2,2)=7, проследив за ростом стека.

в) Попытайтесь вычислить A(4,4) и опишите, что при этом происходит. Попробуйте выполнить вычисление, используя несколько компиляторов. Не забудьте перед этим сохранить свои файлы!

г) Напишите нерекурсивную программу для функции Акерманна.

5. Как получить доступ к переменным промежуточной области действия на процессоре 8086?

6. Существует механизм передачи параметров, называемый вызовом по имени *(call-by-name),* в котором при каждом обращении к формальному параметру происходит перевычисление фактического параметра. Этот механизм впервые использовался в языке Algol, но его нет в большинстве обычных языков программирования. Что послужило причиной такого решения в языке Algol, и как оно было реализовано?

3

Более сложные понятия

Глава 8
Указатели

8.1 . Указательные типы

Переменная — не более чем удобная нотация адресования ячейки памяти. Имя переменной является статическим и определено на этапе компиляции: разные имена относятся к разным ячейкам, и не существует способов «вычисления имени», кроме как в определенных видах контекстов, таких как индексирование массива. Значение *указательного (ссылочного)* типа *(pointer type)* — это адрес; указательная переменная (указатель) содержит адрес другой переменной или константы. Объект, на который указывают, называется *указуемым* или *обозначаемым объектом (designated object).* Указатели применяются скорее для вычислений над адресами ячеек, чем над их содержимым.
Следующий пример:

C

```
int i = 4;
int *ptr = &i;
```

породит структуру, показанную на рис. 8.1. Указатель ptr сам является переменной со своим собственным местом в памяти (284), но его содержимое — это адрес (320) другой переменной i.

Синтаксис объявления может ввести в заблуждение, потому что звездочка «*» по смыслу относится к типу int, а не к переменной ptr.

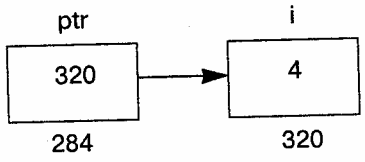

Рис. 8.1. Переменная-указатель и указуемая переменная.

Объявление следует читать как: «ptr имеет указатель типа на int».. Унарная операция «&» возвращает адрес следующего за ней операнда.

К значению переменной i, конечно, можно получить доступ, просто использовав ее имя, например, как i + 1, но к нему также можно получить доступ путем *разыменования (dereferencing)** указателя с помощью синтаксиса *ptr. Когда вы разыменовываете указатель, вы хотите увидеть не содержимое переменной-указателя ptr, а содержимое ячейки памяти, адрес которой содержится в ptr, то есть указуемый объект.

Типизированные указатели

В приведенном примере адреса записаны как целые числа, но адрес *не является* целым числом. Форма записи адреса будет зависеть от архитектуры компьютера. Например, компьютер Intel 8086 использует два 16-разрядных слова, которые объединяются при формировании 20-разрядного адреса. Разумно предположить, что все указатели представляются единообразно.

Однако в программировании полезнее и надежнее использовать типизированные указатели, которые объявляются, чтобы ссылаться на конкретный тип, такой как тип int в приведенном выше примере. Указуемый объект *ptr должен иметь целый тип, и после разыменования его можно использовать в любом контексте, в котором требуется число целого типа:

```
inta[10];
a[*ptr] = a[(*ptr) + 5]; /* Раскрытие и индексирование */
a[i] = 2 * *ptr; /* Раскрытие и умножение */
```

Важно делать различие между переменной-указателем и указуемым объектом и быть очень осторожными при присваивании или сравнении указателей:

```
int i1 = 10;
int i2 = 20;
int *ptr1 = &i1;          /* ptrl указывает на i1 */
int *ptr2 = &i2;          /* ptr2 указывает на i2 */

*ptr1 = *ptr2;            /* Обе переменные имеют одно и то же значение */
if(ptr1 == ptr2)...       /* «Ложь», разные указатели */
if (*ptr1 == *ptr2)       /* «Истина», обозначенные объекты равны */
ptrl = ptr2;              /* Оба указывает на i2 */
```

C

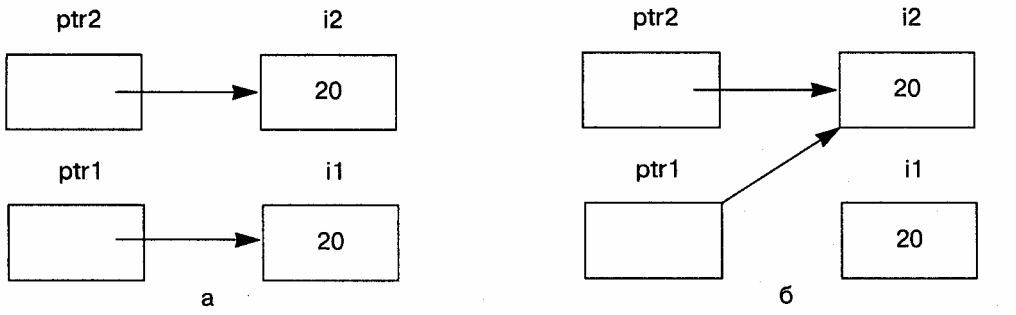

Рис. 8.2. Присваивания с указателями.

На рисунке 8.2а показаны переменные после первого оператора присваивания: благодаря раскрытию указателей происходит присваивание указуемых объектов и i1 получает значение 20. После выполнения второго оператора присваивания (над указателями, а не над указуемыми объектами) переменная i1 больше не является доступной через указатель, что показано на рис. 8.26.

Важно понимать различие между указателем-константой и указателем на константный указуемый объект. Создание указателя-константы не защищает указуемый объект от изменения:

```
inti1,i2;
int * const p1 = &i1;                    /* Указатель-константа */
const int * p2 = &i1;                    /* Указатель на константу */
const int * const p3 = &i1;              /* Указатель-константа на константу */

p1 =&i2;                                 /* Ошибка, указатель-константа */
*p1=5                                    /* Правильно, указуемый объект не является
                                            константой */
p2 =&i2;                                 /* Правильно, указатель не является
                                            константой */
*p2 = 5;                                 /* Ошибка, указуемый объект — константа */
 p3 =&i2;                                /* Ошибка, указатель-константа */
*p3 = 5;                                 /* Ошибка, указуемый объект — константа */
```

В языке С указатель на void является нетипизированным указателем. Любой указатель может быть неявно преобразован в указатель на void и обратно, хотя смешанное использование присваиваний типизированных указателей обычно будет сопровождаться предупреждающим сообщением. К счастью, в C++ контроль соответствия типов делается намного тщательнее. Типизированные указатели неявно могут быть преобразованы в указатели на void, но не обратно:

```
void        *void_ptr;                   /* Нетипизированный указатель */
int         *int_ptr;                    /* Типизированный указатель */
char        *char_ptr;                   /* Типизированный указатель */
void_ptr = int_ptr;                      /* Правильно */
char_ptr = void_ptr;                     /* Правильно в С, но ошибка в C++ */
char_ptr = int_ptr;                      /* Предупреждение в С, ошибка в C++ */
```

C

Поскольку в С нет контроля соответствия типов, указателю может быть присвоено произвольное выражение. Нет никакой гарантии, что указуемый объект имеет ожидаемый тип; фактически значение указателя могло бы даже не быть адресом в отведенной программе области памяти. В лучшем случае это приведет к аварийному сбою программы из-за неправильной адресации, и вы получите соответствующее сообщение от операционной системы. В худшем случае это может привести к разрушению данных операционной

системы. Ошибки в указателях очень трудно выявлять при отладке, потому что сложно разобраться в абсолютных адресах, которые показывает отладчик. Решение состоит в более строгом контроле соответствия типов для указателей, как это делается в Ada и C++.

Синтаксис

Синтаксические конструкции, связанные с указателями, иногда могут вводить в заблуждение, поэтому очень важно хорошо их понимать. Раскрытие указателей, индексация массивов и выбор полей записей — это средства доступа к данным внутри структур данных. В языке Pascal синтаксис самый ясный: каждая из этих трех операций обозначается отдельным символом, который пишется после переменной. В следующем примере Ptr объявлен как указатель на массив записей с целочисленным полем:

Pascal

```
type Rec_Type =
record
    Field: Integer;
end;
type Array_Type = array[1 ..100] of Rec_Type;
type Ptr_Type =  Array_Type;
Ptr: Ptr_Type;
Ptr (*Указатель на массив записей с целочисленным полем *)
Ptrt (*Массив записей с целочисленным полем *)
Ptrt [78] (*Запись с целочисленным полем *)
Ptrt [78].Field ("Целочисленное поле *)
```

В языке C символ раскрытия ссылки (*) является префиксным оператором, поэтому приведенный пример записывался бы так:

C

```
typedef struct {
        int field;
} Rec_Type;
 typedef Rec_Type Array_Type[ 100];

Array_Type *ptr;

ptr                     /* Указатель на массив записей с целочисленным полем */
*ptr                    /* Массив записей с целочисленным полем */
(*ptr)[78]              /* Запись с целочисленным полем */
(*ptr)[78].field        /* Целочисленное поле */
```

Здесь необходимы круглые скобки, потому что индексация массива имеет более высокий приоритет, чем раскрытие указателя. В сложной структуре данных это может внести путаницу при расшифровке декомпозиции, которая использует разыменование как префикс, а индексацию и выбор поля как постфикс. К счастью, наиболее часто используемая последовательность операций, в которой за разыменованием следует выбор поля, имеет специальный, простой синтаксис. Если ptr указывает на запись, то ptr->field — это краткая запись для (*ptr).field.

Синтаксис Ada основан на предположении, что за разыменованием почти всегда следует выбор поля, поэтому отдельная запись для разыменования не нужна. Вы не можете сказать, является R.Field просто выбором поля обычной записи с именем R, или R — это указатель на запись, который раскрывается перед выбором. Хотя такой подход и может привести к путанице, но он имеет то преимущество, что в структурах данных мы можем перейти от использования самих записей к использованию указателей на них без других изменений

программы. В тех случаях, когда необходимо только разыменование, используется довольно неуклюжий синтаксис, как показывает вышеупомянутый пример на языке Ada:

```
type Rec_Type is
    record
        Field: Integer;
    end record;
type Array_Type is array( 1 .. 100) of Rec_Type;
type Ptr_Type is access Array_Type;

Ptr: Ptr_Type;
Ptr                    -- Указатель на массив записей с целочисленным полем
Ptr.all                -- Массив записей с целочисленным полем
Ptr.all[78]            -- Запись с целочисленным полем
Ptr.all[78].Field      --Целочисленное поле
```

Ada

Обратите внимание, что в Ada для обозначения указателей используется ключевое слово access, а не символ. Ключевое слово all используется в тех немногих случаях, когда требуется разыменование без выбора.

Реализация

Для косвенного обращения к данным через указатели требуется дополнительная команда в машинном коде. Давайте сравним прямой оператор присваивания с косвенным присваиванием, например:

```
int i,j;
int*p = &i;
int *q = &j;

i=j;                    /* Прямое присваивание */
*p = *q;                /* Косвенное присваивание */
```

C

Машинные команды для прямого присваивания:

```
load    R1J
store   R1,i
```

C

в то время как команды для косвенного присваивания:

```
load    R1,&q       Адрес (указуемого объекта)
load    R2,(R1)     Загрузить указуемый объект
load    R3,&p       Адрес (указуемого объекта)
store   R2,(R3)     Сохранить в указуемом объекте
```

C

При косвенности неизбежны некоторые издержки, но обычно не серьезные, поскольку при неоднократном обращении к указуемому объекту оптимизатор может гарантировать, что указатель будет загружен только один раз. В операторе

```
p->right = p->left;
```

раз уж адрес p загружен в регистр, все последующие обращения могут воспользоваться этим регистром:

```
load    R1 ,&p          Адрес указуемого объекта
load    R2,left(R1)     Смещение от начала записи
store   R2,right(R1)    Смещение от начала записи
```

Потенциальным источником неэффективности при косвенном доступе к данным через указатели является размер самих указателей. В начале 1970-х годов, когда разрабатывались языки C и Pascal, компьютеры обычно имели только 16 Кбайт или 32 Кбайт оперативной памяти, и для адреса было достаточно 16 разрядов. Теперь, когда персональные компьютеры и рабочие станции имеют много мегабайтов памяти, указатели должны храниться в 32 разрядах. Кроме того, из-за механизмов управления памятью, основанных на кэше и страничной организации, произвольный доступ к данным через указатели может обойтись намного дороже, чем доступ к массивам, которые располагаются в непрерывной последовательности ячеек. Отсюда следует, что оптимизация структуры данных для повышения эффективности сильно зависит от системы, и ее никогда не следует делать до измерения времени выполнения с помощью профилировщика.

Типизированные указатели в Ada предоставляют одну возможность для оптимизации. Для набора указуемых объектов, связанных с конкретным типом доступа, т. е. для так называемой *коллекции (collection),* можно задать размер:

```
type Node_Ptr is access Node;
for Node_Ptr'Storage_Size use 40_000;
```
C

Поскольку объем памяти, запрошенный для Node, меньше 64 Кбайт, указатели относительно начала блока могут храниться в 16 разрядах, при этом экономятся и место в структурах данных, и время центрального процессора для загрузки и сохранения указателей.

Указатели и алиасы в Ада 95

Указатель в языке C может использоваться для задания алиаса (альтернативного имени) обычной переменной:

```
inti;
int *ptr = &i;
```
C

Алиасы бывают полезны; например, они могут использоваться для создания связанных структур во время компиляции. Так как в Ада 83 структуры, основанные на указателях, могут быть созданы только при выполнении, это может привести к ненужным издержкам и по времени, и по памяти.

В Ada 95 добавлены специальные средства создания алиасов, названные *типами обобщенного доступа (general access types),* но на них *наложены* ограничения для предотвращения создания повисших ссылок (см. раздел 8.3). Предусмотрен и специальный синтаксис как для объявления указателя, так и для переменной с алиасом:

```
type Ptr is access all Integer;     -- Ptr может указывать на алиас
I: aliased Integer;                 -- I может иметь алиас
P: Ptr := I'Access;                 -- Создать алиас
```
C

Первая строка объявляет тип, который может указывать на целочисленную переменную с алиасом, вторая строка объявляет такую переменную, и третьястрока объявляет указатель и инициализирует его адресом переменной. Такие типы обобщенного доступа и переменные с алиасом могут быть компонентами массивов и записей, что позволяет построить связанные структуры, не обращаясь к администратору памяти во время выполнения.

* Привязка к памяти

В языке С привязка к памяти тривиальна, потому что указателю может быть присвоен произвольный адрес:

C

```
int * const reg = Ox4fOO;          /* Адрес (в шестнадцатеричной системе) */
*reg = Ox1f1f;                     /* Присваивание по абсолютному адресу */
```

Благодаря использованию указателя-константы мы уверены, что адрес в reg не будет случайно изменен.

В Ada используется понятие спецификации представления для явного установления соответствия между обычной переменной и абсолютным адресом:

Ada

```
Reg: Integer;
for Reg use at 16#4fOO#;           -- Адрес (в шестнадцатеричной системе)
Reg := 16#1 f1 f#;                 -- Присваивание по абсолютному адресу
```

Преимущество метода языка Ada состоит в том, что не используются явные указатели.

8.2. Структуры данных

Указатели нужны для реализации динамических структур данных, таких как списки и деревья. Кроме элементов данных узел в структуре содержит один или несколько указателей со ссылками на другие узлы (см. рис. 8.3).

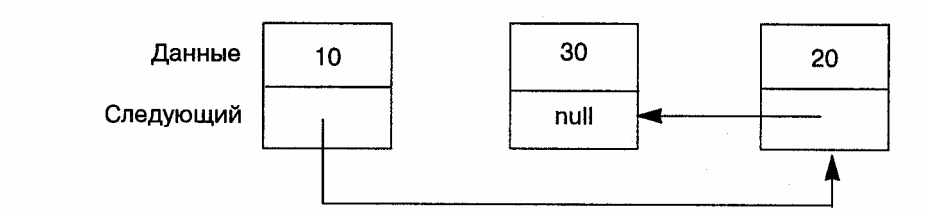

Рис. 8.3. Динамическая структура данных.

Попытка определить узел неизбежно ведет к рекурсии в определении типа, а именно: запись типа node (узел) должна содержать указатель на свойсобственный тип node. Для решения этой проблемы в языках допускается задавать частичное объявление записи, в котором указывается имя ее типа. Объявление сопровождается объявлением указателя, ссылающегося на это имя, а далее следует полное объявление записи, в котором уже можно ссылаться на тип указателя. В языке Ada эти три объявления выглядят так:

Ada

```
type Node;                         -- Незавершенное объявление типа
type Ptr is access Node;           -- Объявление типа указателя
type Node is                       -- Полное объявление
    record
      Data: Integer;               -- Данные в узле
      Next: Ptr;                   -- Указатель на следующий узел
    end record;
```

Язык C требует использования *тега* структуры и альтернативного синтаксиса для объявления записи:

```
typedef struct node *Ptr;                    /* Указатель на структуру с тегом */
typedef struct node {                        /* Объявление структуры узла*/
        int      data;                       /* Данные в узле */
        Ptr      next;                       /* Указатель на следующий узел */
} node;
```

C

В C++ нет необходимости использовать typedef, поскольку struct определяет как тег структуры, так и имя типа:

```
typedef struct node *Ptr;                    /* Указатель на структуру с тегом */
struct node {                                /* Объявление структуры узла */
int       data;                              /* Данные в узле */
Ptr        next;                             /* Указатель на следующий узел */
}
```

C++

Алгоритмы для *прохождения (traverse)* структур данных используют переменные-указатели. Следующий оператор в C — это поиск узла, поле данных которого содержит key:

```
while (current->data != key)
     current = current->next;
```

C

Аналогичный оператор в Ada (использующий неявное раскрытие ссылки) таков:

```
while Current.Data /= Key loop
      Current := Current.Next;
end loop;
```

Ada

Структуры данных характеризуются числом указателей, хранящихся в каждом узле, тем, куда они указывают, и алгоритмами, используемыми для прохождения структур и их обработки. Все алгоритмы, излагаемые в учебных курсах по структурам данных, достаточно просто программируются на языках C или Ada с использованием записей и указателей.

Указатель null (пустой)

На рисунке 8.3 поле next последнего элемента списка не указывает ни на что. Обычно считается, что такой указатель имеет специальное значение — *пустое*, которое отличается от любого допустимого указателя. Пустое значение в Ada обозначается зарезервированным словом null. В предыдущем разделе, чтобы не пропустить конец списка, поиск фактически следовало бы запрограммировать следующим образом:

```
while (Current /= null) and then (Current.Data /= Key) loop
       Current := Current.Next;
end loop;
```

Ada

Обратите внимание, что укороченное вычисление (см. раздел 6.2) здесь существенно.

В языке C используется обычный целочисленный литерал «ноль» для обозначения пустого указателя:

```
while ((current != 0) && (current->data != key))
current = current->next;
```

C

Нулевой литерал — это всего лишь синтаксическое соглашение; реальное значение зависит от компьютера. При просмотре с помощью отладчика в пустом указателе *все биты могут быть, а могут и не быть нулевыми.* Для улучшения читаемости программы в библиотеке C определен символ NULL:

```
while ((current != NULL) && (current->data != key))
current = current->next;
```
C

Когда объявляется переменная, например целая, ее значение не определено. И это не вызывает особых проблем, поскольку любая комбинация битов задает допустимое целое число. Однако указатели, которые не являются пустыми и при этом не ссылаются на допустимые блоки памяти, могут вызвать серьезные ошибки. Поэтому в Ada каждая переменная-указатель неявно инициализируется как null. В языке C каждая *глобальная* переменная неявно инициализируется как ноль; глобальные переменные-указатели инициализируются как пустые. Позаботиться о явной инициализации локальных указателей должны вы сами.

Нужно быть очень осторожными, чтобы случайно не разыменовать пустой указатель, потому что значение null не указывает ни на что (или, вернее, ссылается на данные системы по нулевому адресу):

```
Current: Ptr := null;
Current := Current.Next;
```
Ada

В языке Ada эта ошибка будет причиной исключительной ситуации (см. гл. 11), но в C результат попытки разыменовывать null может привести к катастрофе. Операционные системы, которые защищают программы друг от друга, смогут прервать «провинившуюся» программу; без такой защиты разыменование могло бы вмешаться в другую программу или даже разрушить систему.

Указатели на подпрограммы

В языке C указатель может ссылаться на функцию. При программировании это чрезвычайно полезно в двух случаях:

• при передаче функции как параметра,

• при создании структуры данных, которая каждому ключу или индексу ставит в соответствие процедуру.

Например, один из параметров пакета численного интегрирования — это функция, которую нужно проинтегрировать. Это легко запрограммировать в C, создавая тип данных, который является указателем на функцию; функция получит параметр типа float и вернет значение типа float:

```
typedef float (*Func) (float);
```
C

Этот синтаксис довольно плох потому, что имя типа (в данном случае — Func) находится глубоко внутри объявления, и потому, что старшинство операций в C требует дополнительных круглых скобок.

Раз тип объявлен, он может использоваться как тип формального параметра:

```
float integrate(Func f, float upper, float lower)
{
float u = f (upper); float I = f(lower);
}
```
C

Обратите внимание, что раскрытие указателя делается автоматически, когда вызывается функция-параметр, иначе нам пришлось бы написать (*f)(upper). Теперь, если определена функция с соответствующей сигнатурой, ее можно использовать как фактический параметр для подпрограммы интегрирования:

```
float fun (float parm)
{
…                                   /* Определение "fun" */
}
float x = integrate(fun, 1.0, 2.0);          /* "fun" как фактический параметр */
```
C

Структуры данных с указателями на функции используются при создании *интерпретаторов* — программ, которые получают последовательность кодов и выполняют действия в соответствии с этими кодами. В то время как статический интерпретатор может быть реализован с помощью case-оператора и обычных вызовов процедур, в динамическом интерпретаторе соответствие между кодами и операциями будет устанавливаться только во время выполнения. Современные системы с окнами используют аналогичную методику программирования: программист должен предоставить возможность *обратного вызова (callback),* т.е. процедуру, обеспечивающую выполнение соответствующего действия для каждого события. Это указатель на подпрограмму, которая будет выполнена, когда получен код, указывающий, что событие произошло:

```
typedef enum {Event1, ..., Event'10} Events;
typedef void (*Actions)(void);
                /* Указатель на процедуру */
Actions action [10];
                 /* Массив указателей на процедуры */
```
C

Во время выполнения вызывается процедура, которая устанавливает соответствие между событием и действием:

```
void insta!l(Events e, Actions a)
{
action[e] = a;
}
```
C

Затем, когда событие происходит, его код может использоваться для индексации и вызова соответствующей подпрограммы:

```
action [e] ();
```
C

Поскольку в Ada 83 нет указателей на подпрограммы, эту технологию нельзя запрограммировать без использования нестандартных средств. Когда язык разрабатывался, указатели на подпрограммы были опущены, потому что предполагалось, что родовых (generics)* программных модулей (см. раздел 10.3) будет достаточно для создания математических библиотек, а методика обратного вызова еще не была популярна. В Ada 95 этот недостаток устранен, и разрешены указатели на подпрограммы. Объявление математической библиотечной функции таково:

```
type Func is access function(X: Float) return Float;
                    -- Тип: указатель на функцию
function lntegrate(F: Func; Upper, Lower: Float);
                    -- Параметр является указателем на функцию
```
Ada

а обратный вызов объявляется следующим образом:

```
type Events is (Event'1,..., EventIO);
type Actions is access procedure;
            -- Тип: указатель на процедуру
Action: array(Events) of Actions;
            -- Массив указателей на процедуры
```
Ada

Указатели и массивы

В языке Ada в рамках строгого контроля типов единственно допустимые операции на указателях — это присваивание, равенство и разыменование. В языке C, однако, считается, что указатели будут неявными последовательными адресами, и допустимы арифметические операции над значениями указателей. Это ясно из взаимоотношений указателей и массивов: указатели рассматриваются как более простое понятие, а доступ к массиву определяется в терминах указателей. В следующем примере

```
int *ptr;                   /* Указатель на целое */
int a[100];                 /* Массив целых чисел */
ptr = &a[0];                /* Явный адрес первого элемента
*/ ptr = a;                 /* Неявный тот же адрес */
```
C

два оператора присваивания эквивалентны, потому что имя массива рассматривается всего лишь как указатель на первый элемент массива. Более того, если прибавление или вычитание единицы делается для указателя, результат будет не числом, а результатом увеличения или уменьшения указателя на размер типа, на который ссылается указатель. Если для целого числа требуются четыре байта, а p содержит адрес 344, то p+1 равно не 345, а 348, т.е. адресу «следующего» целого числа. Доступ к элементу массива осуществляется прибавлением индекса к указателю и разыменованием, следовательно, два следующих выражения эквивалентны:

```
*(ptr + i)
a[i]
```
C

Несмотря на эту эквивалентность, в языке C все же остается значительное различие между массивом и указателем:

```
char s1[] = "Hello world";
char *s2 = "Hello world";
```
C

Здесь s1 — это место расположения последовательности из 12 байтов, содержащей строку, в то время как s2 — это переменная-указатель, содержащая адрес аналогичной последовательности байтов (см. рис. 8.4). Однако s1[i] —это то же самое, что и *(s2+i) для любого i из рассматриваемого диапазона, потому что массив при использовании автоматически преобразуется в указатель.

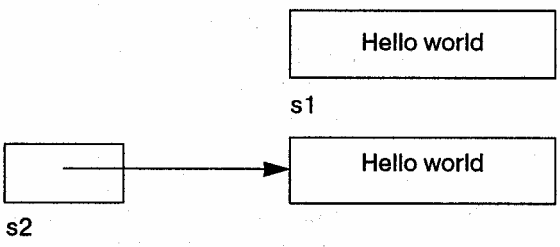

Рис. 8.4. Массив и указатель в языке С.

Проблема арифметических операций над указателями состоит в том, что нет никакой гарантии, что результат выражения действительно ссылается на элемент массива. Тогда как нотацию индексации относительно легко понять и быть уверенным в ее правильности, арифметических операций над указателями по возможности следует избегать. Однако они могут быть очень полезны для улучшения эффективности в циклах, если ваш оптимизатор недостаточно хорош.

8.3. Распределение памяти

При выполнении программы память используется для хранения как программ (кода), так и различных структур данных, например стека. Хотя распределение и освобождение памяти правильнее обсуждать в контексте компиляторов и операционных систем, вполне уместно сделать обзор этой темы здесь, потому что реализация может существенно повлиять на выбор конструкций языка и стиля программирования.

Существует пять типов памяти, которые должны быть выделены.

Код. Машинные команды, которые являются результатом компиляции программы.

Константы. Небольшие константы, такие как 2 и 'x', часто могут содержаться внутри команды, но для больших констант память должна выделяться особо, в частности для констант с плавающей точкой и строк.

Стек. Стековая память используется в основном для записей активации, которые содержат параметры, переменные и ссылки. Она также используется для временных переменных при вычислении выражений.

Статические данные. Это переменные, объявленные в главной программе и в других местах: в Ada — данные, объявленные непосредственно внутри библиотечных пакетов; в С — данные, объявленные непосредственно внутри файла или объявленные как статические (static) в блоке.

Динамическая область. Динамическая область (куча — *heap)* — термин, используемый для области данных, из которой данные динамически выделяются командой malloc в С и new в Ada и C++.

Код и константы похожи тем, что они определяются во время компиляции и уже не изменяются. Поэтому в дальнейшем обсуждении мы объединим эти два типа памяти вместе. Обратите внимание, что, если система это поддерживает, код и константы могут храниться в памяти, доступной только для чтения (ROM). Стек обсуждался подробно в разделе 7.6.

Мы упомянули, что статические (глобальные) данные можно считать распределенными в начале стека. Однако статические данные обычно распределяются независимо. Например, в Intel 8086 каждая область данных (называемая *сегментом)* ограничена 64 Кбайтами. Поэтому есть смысл выделять отдельный сегмент для стека помимо одного или нескольких сегментов для статических данных.

И наконец, мы должны выделить память для кучи. Динамическая область отличается от стека тем, что выделение и освобождение памяти может быть очень хаотичным. Исполняющая система должна применять сложные алгоритмы, чтобы гарантировать оптимальное использование динамической области.

Программа обычно помещается в отдельную, непрерывную область. Память должна быть разделена так, чтобы разместить требуемые области памяти. На рисунке 8.5 показано, как это реализуется. Поскольку области кода, констант и статических данных имеют фиксированные размеры, они распределяются в начале памяти. Две области переменной длины, куча и стек помещаются в противоположные концы остающейся памяти.

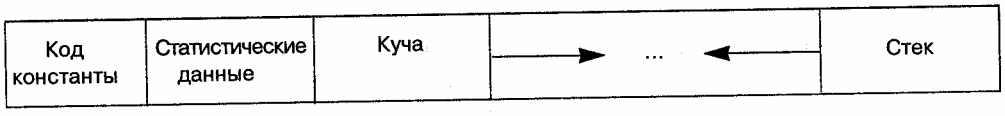

Рис. 8.5. Распределение памяти: код, данные, стек и куча.

При таком способе, если программа использует большой стек во время одной фазы вычисления и большую кучу во время другой фазы, то меньше шансов, что памяти окажется недостаточно.

Важно понять, что каждое выделение памяти в стеке или в куче (то есть каждый вызов процедуры и каждое выполнение программы выделения памяти) может закончиться неудачей из-за недостатка памяти. Тщательно разработанная программа должна уметь восстанавливаться при недостатке памяти, но такую ситуацию нелегко обработать, потому что процедуре, которая выполняет восстановление, может понадобиться еще больший объем памяти! Поэтому желательно получать сигнал о недостатке памяти, когда еще остается значительный резерв.

Запрос и освобождение памяти

В процедурных языках программирования есть явные выражения или операторы запроса и освобождения памяти. Язык C использует malloc, функцию весьма опасную, поскольку в ней никак не проверяется соответствие выделенного объема памяти размеру указуемого объекта. Следует использовать функцию sizeof, даже когда это явно не требуется:

C

```
int*p = (int*)malloc(1);                    /* Ошибка */
int *p = (int *) malloc(sizeof(int));       /* Этот вариант лучше */
```

Обратите внимание, что malloc возвращает *нетипизированный* указатель, который должен быть явно преобразован к требуемому типу.

При освобождении памяти задавать размер блока не нужно:

```
free(p);
```

Выделенный блок памяти включает несколько дополнительных слов, которые используются для хранения размера блока." Этот размер используется в алгоритмах управления динамической областью, как описано ниже.

Языки C++ и Ada используют нотацию, из которой ясно видно, что создается указуемый объект конкретного типа. При этом нет опасности несовместимости типа и размера объекта:

```
typedef Node *Node_Ptr;
Node_Ptr *p = new Node;                    // C++

type Node_Ptr is access Node;
P: Node_Ptr := new Node;                    --Ada
```

Оператор delete освобождает память в C++. Ada предпочитает, чтобы вы не освобождали память, выделенную в куче, потому что освобождение памяти опасно по существу (см. ниже). Конечно, на практике без освобождения не обойтись, поэтому применяемый метод назван *освобождением без контроля (unchecked deallocation),* и назван он так для напоминания, что его использование опасно. Обратите внимание, что освобождаемая память — это область хранения указуемого объекта (на который *ссылается* указатель), а не самого указателя.

Повисшие ссылки

Серьезная опасность, связанная с указателями, — это возможность создания *повисших ссылок (danglingpointers)* при освобождении блока памяти:

C++

```
int *ptr1 = new int; int *ptr2;
ptr2 = ptrl;                    // Оба указывают на один и тот же блок
result = delete ptrl;           // ptr2 теперь указывает на освобожденный блок
```

После выполнения первого присваивания оба указателя ссылаются на выделенный блок памяти. Когда память освобождена, второй указатель все еще сохраняет копию адреса, но этот адрес теперь не имеет смысла. В алгоритме со сложной структурой данных нетрудно создать двойную ссылку такого рода по ошибке.

Повисшие ссылки могут возникать также в C и C++ без какого-либо явного участия программиста в освобождении памяти:

C

```
char *proc(int i)                   /* Возвращает указатель на тип char */
{
        char c;                     /* Локальная переменная */
        return &c;                  /* Указатель на локальную переменную типа
                                        char */
}
```

Память для с неявно выделяется в стеке при вызыве процедуры и неявно освобождается после возврата из процедуры, поэтому возвращенное значение указателя больше не ссылается на допустимый объект. Это легко увидеть в процедуре из двух строк, но, возможно, не так легко заметить в большой программе.

Ada пытается избежать повисших ссылок.

• Указатели на объекты (именованные переменные, константы и параметры) запрещены в Ada 83; в Ada 95 они вводятся специальной конструкцией alias, правила которой предотвращают возникновение повисших ссылок.

• Явного выделения памяти избежать нельзя, поэтому применяемый метод назван Unchecked Deallocation (освобождение без контроля) с целью предупредить программиста об опасности.

8.4. Алгоритмы распределения динамической памяти

Менеджер кучи — это компонент исполняющей системы, который выделяет и освобождает память. Это делается посредством поддержки списка *свободных блоков*. Когда сделан запрос на выделение памяти, она ищется в этом списке, а при освобождении блок снова подсоединяется к списку свободных блоков. Разработчик исполняющей системы должен рассмотреть много вариантов и принять проектные решения, в частности по порядку обработки блоков, их структуре, порядку поиска и т. д.

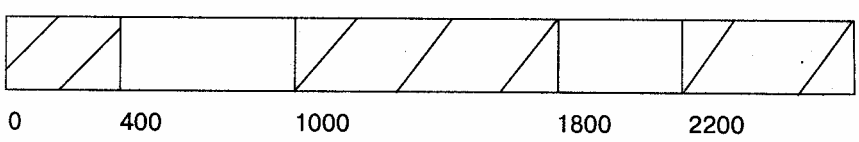

Рис. 8.6. Фрагментация динамической памяти.

.

С распределением динамической области памяти связана проблема фрагментации. На рисунке 8.6 показана ситуация, когда сначала были выделены пять блоков памяти, а затем второй и четвертый освобождены. Теперь, хотя доступны 1000 байтов, невозможно выделить больше 600 байтов, потому что память раздроблена на небольшие блоки. Даже когда третий блок освободится, памяти будет достаточно только при условии, что менеджер кучи «умеет» сливать смежные свободные блоки.

В добавление к слияниям менеджер кучи может предупреждать фрагментацию, отыскивая блок подходящего размера, а не просто первый доступный, или выделяя большие блоки из одной области динамической памяти, а небольшие блоки — из другой. Существует очевидный компромисс между сложностью менеджера и издержками времени выполнения.

Программист должен знать используемые алгоритмы управления динамической памятью и писать программу с учетом этих знаний.

Другая возможность ослабить зависимость от алгоритмов работы менеджера кучи — это завести кэш освобождаемых блоков. Когда блок освобождается, он просто подсоединяется к кэшу. Когда необходимо выделить блок, сначала проверяется кэш; это позволяет избежать издержек и фрагментации, возникающих при обращениях к менеджеру кучи.

В Ada есть средство, которое позволяет программисту задать несколько куч разного размера, по одной для каждого типа указателя. Это позволяет предотвратить фрагментацию, но повышает вероятность того, что в одной куче память будет исчерпана, в то время как в других останется много свободных блоков.

Виртуальная память

Есть один случай, когда распределение динамической памяти совершенно надежно — это когда используется *виртуальная память*. В системе с виртуальной памятью программисту предоставляется настолько большое адресное пространство, что переполнение памяти фактически невозможно. Операционная система берет на себя распределение логического адресного пространства в физической памяти, когда в этом возникает необходимость. Когда физическая память исчерпана, блоки памяти, называемые *страницами*, выталкиваются на диск.

С помощью виртуальной памяти менеджер кучи может продолжать выделение динамической памяти почти бесконечно, не сталкиваясь с проблемой фрагментации. Единственный риск — это связанная с виртуальной памятью ситуация пробуксовки *(thrashing)*, которая происходит, когда код и данные, требуемые для фазы вычисления, занимают так много страниц, что в памяти для них не хватает места. На подкачку страниц тратится так много времени, что вычисление почти не продвигается.

Сборка мусора

Последняя проблема, связанная с динамической памятью, — образование мусора *(garbage),* например:

```
int *ptr1 = new int;          // Выделить первый блок
int *ptr2 = new int;          // Выделить второй блок
ptr2 = ptrl;                  // Второй блок теперь недоступен
```

C

После оператора присваивания второй блок памяти доступен через любой из указателей, но нет никакого способа обратиться к первому блоку (см. рис. 8.7). Это может и не быть ошибкой, потому что память, к которой нельзя обратиться, (называемая мусором) не может вам помешать. Однако, если продолжается *утечка памяти,* т. е. образуется мусор, в конечном счете программа выйдет из строя из-за недостатка памяти. Чрезвычайно трудно локализовать причину утечки памяти, потому что нет прямой связи между причиной и симптомом (недостатком памяти).

Очевидное решение состоит в том, чтобы не создавать мусор, прежде всего тщательно заботясь об освобождении каждого блока до того, как он станет недоступен. Кроме того, исполняющая система языка программирования может содержать *сборщик мусора (garbage collector).* Задача сборщика мусора состоит в том, чтобы «повторно использовать» мусор, идентифицируя недоступные блоки памяти и возвращая их менеджеру динамической памяти. Существует два основных алгоритма сборки мусора: один из них для каждого блока

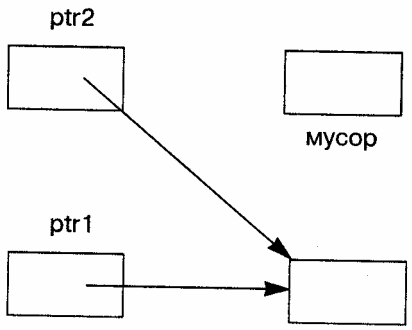

Рис. 8.7. Образование мусора.

ведет счетчик текущего числа указателей, ссылающихся на этот блок, и автоматически освобождает блок, когда счетчик доходит до нуля. Другой алгоритм отмечает все доступные блоки и затем собирает немаркированные (и, следовательно, недоступные) блоки. Первый алгоритм проблематичен, потому что группа блоков, каждый из которых является мусором, могут указывать друг на друга так, что счетчик никогда не сможет уменьшиться до нуля. Второй алгоритм требует прерывания вычислений на длительные периоды времени, чтобы маркировку и сбор можно было выполнить без влияния вычислений. Это, конечно, недопустимо в интерактивных системах.

Сборка мусора традиционно выполняется в таких языках, как Lisp и Icon, которые создают большое число временных структур данных, быстро становящихся мусором. Проведены обширные исследования по сборке мусора; особое внимание в них уделено параллельным и пошаговым методам, которые не будут нарушать интерактивные вычисления или вычисления в реальном масштабе времени. Eiffel — один из немногих процедурных языков, которые включают сборщики мусора в свои исполняющие системы.

8.5. Упражнения

1. Как представлен на вашем компьютере указатель? Как представлен на вашем компьютере указатель null?

2. Напишите на языке С алгоритм обработки массива с помощью индексации, а затем измените его, чтобы использовать явные операции с указателями. Сравните получающиеся в результате машинные команды и время выполнения двух программ. Есть ли различие в оптимизации?

3. Покажите, как можно применить «часовых», чтобы сделать поиск в списке более эффективным.

4. Почему не была использована операция адресации для фактического параметра, являющегося указателем на функцию:

```
float x = integrate(&fun, 1.0, 2.0);
```

C

5. Покажите, как можно использовать повисшие ссылки, чтобы разрушить систему типов.

6. Изучите в Ada 95 определение *доступности (accessibility)* и покажите, как правила предотвращают возникновение повисших ссылок.

7. Напишите программу обработки динамической структуры данных, например связанного списка. Измените программу, чтобы использовать кэш узлов.

8. Изучите документацию вашего компилятора; с помощью каких алгоритмов исполняющая система распределяет динамическую память? Есть ли какие-либо издержки по памяти при выделении динамической памяти, т. е. выделяются ли лишние слова кроме тех, которые вы запросили? Если да, то сколько?

9. Если у вас есть доступ к компьютеру, который использует виртуальную память, посмотрите, как долго можно продолжать запрашивать память. При нарушении каких пределов выделение памяти прекращается?

Глава 9
Вещественные числа

9.1. Представление вещественных чисел

В главе 4 мы обсуждали, как целочисленные типы используются для представления подмножества математических целых чисел. Вычисления с целочисленными типами могут быть причиной переполнения — это понятие не име-ет никакого смысла для математических целых чисел — а возможность пере-полнения означает, что коммутативность и ассоциативность арифметических
операций при машинных вычислениях не гарантируются.

Представление вещественных чисел в компьютерах и вычисления с этими представлениями чрезвычайно проблематичны — до такой степени, что при создании важных программ полезно консультироваться со специалистами. В этой главе будут изучены основные понятия, связанные с использованием ве- щественных чисел в вычислениях; чрезвычайная легкость написания в про-грамме вычислений с вещественными числами не должна заслонять глубин-ные проблемы.

Прежде всего обратим внимание на то, что десятичные числа не всегда можно точно представить в двоичной нотации. Например, нельзя точно пред-ставить в виде двоичного числа 0.2 (одну пятую), а только как периодическую I двоичную дробь:

0.0011001100110011..

Существуют два решения этой проблемы:

• Представлять непосредственно десятичные числа, например, каждому десятичному символу ставить в соответствие четыре бита. Такое представление называется *двоично-кодированным десятичным числом* (BCD — *binary-coded decimal).*

• Хранить двоичные числа и принять как факт то, что некоторая потеря точности иногда может случаться.

Представление BCD приводит к некоторому перерасходу памяти, потому что с помощью четырех битов можно представить 16 разных значений, а не 10, необходимых для представления десятичных чисел. Более существенный не-достаток состоит в том, что это представление не «естественно», и вычисление с BCD выполняется намного медленнее, чем с двоичными числами. Таким образом, мы ограничимся обсуждением двоичных представлений; читателя, интересующегося вычислениями с BCD, можно отослать к таким языкам, как Cobol, которые поддерживают числа BCD.

Числа с фиксированной точкой

Для простоты последующее обсуждение будет вестись в терминах десятичных чисел, но оно справедливо и для двоичных. Предположим, что мы можем представить в 32-разрядном слове памяти семь цифр: пять до и две после десятичной точки:

12345.67, -1234.56, 0.12

Такое представление называется представлением *с фиксированной точкой*. Преимущество чисел с фиксированной точкой состоит в том, что *количество знаков после запятой*, которое определяет *абсолютную ошибку*, фиксировано. Если перечисленные выше числа обозначают доллары и центы, то любая ошибка, вызванная ограниченным размером слова памяти, не превышает одного цента. Недостаток же состоит в том, что *точность представления*, то есть *относительная ошибка*, которая определяется числом значащих цифр, является переменной. Первое число использует все семь цифр представления, имеющихся в распоряжении, тогда как последнее число использует только две цифры. Хуже то, что переменная точность представления означает, что многие важные числа, такие как сумма $1532 854.07, которую вы выиграли в лотерее, или размер $0.00572 вашего подоходного налога, вообще никак нельзя представить.

Числа с фиксированной точкой используются в приложениях, где существенна абсолютная ошибка в конечном результате. Например, бюджетные вычисления обычно делаются с фиксированной точкой, так как требуемая точность представления известна заранее (скажем, 12 или 16 цифр), а бюджет должен быть сбалансирован до последнего цента. Числа с фиксированной точкой также используются в системах управления, где для взаимодействия датчиков и силовых приводов с компьютером используются слова или поля фиксированной длины. Например, скорость можно представить 10-битовым полем с диапазоном значений от 0 до 102.3 км/час; один бит будет представлять 0.1 км/час.

Числа с плавающей точкой

Ученые, которым приходится иметь дело с широким диапазоном чисел, часто используют так называемую *научную нотацию*

123.45×10^3, 1.2345×10^8, -0.00012345×10^7 12345000.0×10^4

Как можно использовать эту нотацию на компьютере? Сначала обратите внимание на то, что здесь присутствуют три элемента информации, которые должны быть представлены: *знак*, *мантисса* (123.45 в первом числе) и экспонента.

На первый взгляд кажется, что нет никакого преимущества в представлении чисел в научной нотации, потому что для представления мантиссы нужна разная точность: пять цифр в первом и втором числах и по восемь цифр для двух других чисел.

Однако, как можно заметить, конечные нулевые цифры мантиссы, большей 1.0 (и ведущие нулевые цифры мантиссы, меньшей 1.0), можно отбросить, если изменить значение (не точность!) экспоненты. Другими словами, мантиссу можно неоднократно умножать или делить на 10 до тех пор, пока она находится в форме, которая использует максимальную точность представления; при каждой такой операции экспонента будет уменьшаться или увеличиваться на 1 соответственно. Например, последние два числа можно записать с помощью мантиссы из пяти цифр:

-0.12345×10^4 0.12345×10^{12}

Для вычислений на компьютере удобно, когда числа представляются в такой стандартной форме, называемой *нормализованной*, в которой первая ненулевая цифра является разрядом десятых долей числа. Это также позволяет сэкономить место в представлении, поскольку десятичная точка всегда находится в одной и той же позиции, и ее не нужно представлять явно. Представление называется *с плавающей точкой*, потому что десятичная точка «плавает» влево или вправо до тех пор, пока число не будет представлено с максимальной точностью.

В чем основной недостаток вычислений, использующих числа с плавающей точкой? Рассмотрим число 0.12345 x 10'°, которое является нормализованной формой с плавающей точкой для числа

1 234 500 000

и предположим, что таким образом банк представил ваш депозит в размере

$1 234 567 890

Управляющий банком был бы горд тем, что *относительная ошибка:*

$$\frac{67\ 890}{1\ 234\ 567\ 890}$$

является очень малой долей процента, но вы оправданно потребовали бы ваши $67 890, которые составляют *абсолютную ошибку*.

Однако в научных вычислениях относительная ошибка намного важнее абсолютной погрешности. В программе, которая контролирует скорость ра-кеты, требование может состоять в том, чтобы ошибка не превышала 0,5%, Хотя это составляет несколько километров в час во время запуска, и несколь-ко сотен километров в час при приближении к орбите. Вычисления с плавающей точкой используются гораздо чаще, чем с фиксированной точкой, пото-му что относительная точность требуется намного чаще, чем абсолютная. По Этой причине в большинстве компьютеров есть аппаратные средства, которые Непосредственно реализуют вычисления с плавающей точкой.

Представление чисел с плавающей точкой

Числа с плавающей точкой хранятся как *двоичные* числа в нормализованной форме, которую мы описали:

$$-0.101100111 \times 2^{15}$$

При типичной реализации на 32-разрядном компьютере 1 бит выделяется для знака, 23 бита — для мантиссы и 8 битов — для экспоненты. Поскольку для хранения одной десятичной цифры требуется $\log_2 10 = 3.3$ бита, то точность представления составит 23/3.3 = 7 цифр. Если необходима большая точность, то с помощью 64-разрядного двойного слова с 52-разрядной мантиссой можно получить приблизительно 15 цифр точности представления.

Существует «трюк», с помощью которого можно увеличить количество представимых чисел. Так как все числа с плавающей точкой нормализованы и первая цифра нормализованной мантиссы обязательно 1, эту первую цифру можно не представлять явно.

Экспонента со знаком представляется со смещением так, чтобы представление было всегда положительным, и помещается в старшие разряды слова после знакового бита. Это позволяет упростить сравнения, потому что можно воспользоваться обычными целочисленными сравнениямии не выделять специально поля экспоненты со знаком. Например, 8-разрядное поле экспоненты со значениями в диапазоне 0 .. 255 представляет экспоненты в диапазоне -127 .. 128 со смещением 127.

Мы можем теперь расшифровать битовую строку как число с плавающей точкой. Строка

1 1000 1000 0110 0000 0000 0000 0000 000

расшифровывается следующим образом.

• Знаковый бит равен 1, поэтому число отрицательное.

• Представление экспоненты равно 1000 1000 = 128 + 8 = 136. Удаление смещения дает

136-127 = 9

• Мантисса равна 0.10110 ... (обратите внимание, что восстановлен скрытый бит), т. е.

1/2+1/8+.1/16 = 11/16

• Таким образом, хранимое число равно 2^9 x 11/16 = 352.
Как и для целых чисел, для чисел с плавающей точкой переполнение *(overflow)* происходит, когда результат вычисления слишком большой:

$$(0.5x2^{™}) \bullet (0.5 \times 2^{80}) = 0.25 \times 2^{150}$$

Так как самая большая экспонента, которая может быть представлена, равна 128, происходит переполнение. Рассмотрим теперь вычисление:

$$(0.5 \times 2^{-70}) \bullet (0.5 \times 2^{-80}) = 0.25 \times 2^{-150}$$

Говорят, что при вычислении происходит *потеря значимости (underflow),* когда результат слишком мал, чтобы его можно было представить. Вы можете воскликнуть, что такое число настолько мало, что его можно принять равным нулю, и компьютер может интерпретировать потерю значимости именно так, но на самом деле потеря значимости часто говорит об ошибке, которая требует обработки или объяснения.

9.2. Языковая поддержка вещественных чисел

Все языки программирования имеют поддержку вычислений с плавающей точкой. Переменная может быть объявлена с типом float, а литералы с плавающей точкой представлены в форме, близкой к научной нотации:

```
float f1 =7.456;
float f2 = -46.64E-3;
```
C

Обратите внимание, что литералы не нужно представлять в двоичной записи или в нормализованной форме; это преобразование делается компилятором.

Для осмысленных вычислений с плавающей точкой необходимо минимум 32 разряда. Однако часто такой точности недостаточно, поэтому языки поддерживают объявления и вычисления с более высокой точностью. Как минимум, поддерживаются переменные с *двойной точностью (double-precision),* использующие 64 разряда, а некоторые компьютеры или компиляторы поддерживают даже более длинные типы. Двойная точность типов с плавающей точкой называется double в языке C и Long_Float в Ada.

Запись литералов с двойной точностью может быть разной в различных языках. Fortran использует специальную запись, заменяя E, предшествующее экспоненте, на D: -45.64D - 3. В языке C *каждый* литерал хранится с двойной точностью, если же вы хотите задать

одинарную точность, то используется суффикс F. Обратите на это внимание, если вы храните большой массив констант с плавающей точкой.

Ada вводит новое понятие — *универсальные типы (universal types)* — для обработки переменной точности представления литералов. Такой литерал как 0.2 хранится компилятором с потенциально неограниченной точностью (вспомните, что 0.2 нельзя точно представить как двоичное число). Фактически при использовании литерала он преобразуется в константу с той точностью, которая нужна:

Ada

```
PI_F:     constant Float          := 3.1415926535;
PI_L:     constant Long_Float     :=3.1415926535;
PI:       constant                := 3.1415926535;
F: Float          := PI;          -- Преобразовать число к типу Float
L: Long_Float     := PI;          -- Преобразовать число к типу Long_Float
```

В первых двух строках объявляются константы именованных типов. Третье объявление для PI называется *именованным числом (named number)* и имеет универсальный вещественный тип. Фактически, в инициализациях PI преобразуется к нужной точности.

Четыре арифметические операции (+,-,* и /), так же как и операции отношения, определены для типов с плавающей точкой. Такие математические функции, как тригонометрические, могут быть определены в рамках языка (Fortran и Pascal) или поставляться с библиотеками подпрограмм (C и Ada).

Плавающая точка и переносимость

При переносе программ, использующих плавающую точку, могут возникнуть трудности из-за различий в определении спецификаторов типа. Ничто не мешает компилятору для C или Ada использовать 64 разряда для представления float (Float) и 128 разрядов для представления double (Long_Float). Перенос на другую машину проблематичен в обоих направлениях. При переносе с машины, где реализовано представление float с высокой точностью на машину, использующую представление с низкой точностью, все типы float должны быть преобразованы в double, чтобы сохранить тот же самый уровень точности. При переносе с машины с низкой точностью на машину с высокой точностью может потребоваться противоположное изменение, потому что выполнение с избыточной точностью приводит к потерям времени и памяти.

Простейшее частное решение состоит в том, чтобы объявлять и использовать искусственный тип с плавающей точкой; в этом случае при переносе программы нужно будет изменить только несколько строк:

```
typedef double Real;              /* C */
subtype Real is Long_Float;       -- Ada
```

Решение проблемы переносимых вычислений с вещественными числами в Ada
см. в разделе 9.4.

Аппаратная и программная плавающая точка

Наше обсуждение представления чисел с плавающей точкой должно было прояснить, что арифметика на этих значениях является сложной задачей. Нужно разбить слова на составные части, удалить смещение экспоненты, выполнить арифметические операции с несколькими словами, нормализовать результат и представить его как составное слово. Большинство компьютеров использует специальные аппаратные средства для эффективного выполнения вычислений с плавающей точкой.

Компьютер без соответствующих аппаратных средств может все же выполнять вычисления с плавающей точкой, используя библиотеку подпрограмм, которые *эмулируют (emulate)* команды с плавающей точкой. Попытка выполнить команду с плавающей точкой вызовет прерывание «несуществующая команда», которое будет обработано с помощью вызова соответствующей подпрограммы эмуляции. Само собой разумеется, что это может быть очень неэффективно, поскольку существуют издержки на прерывание и вызов под-программы, не говоря о самом вычислении с плавающей точкой.

Если вы предполагаете, что ваша программа будет активно использоваться на компьютерах без аппаратной поддержки плавающей точки, может быть разумнее совсем ей не пользоваться и явно запрограммировать вычисления с фиксированной точкой. Например, финансовая программа может делать все вычисления в центах вместо долей доллара. Конечно, при этом возникает риск переполнения, если типы Integer или Long_integer не представлены с до- статочной точностью.

Смешанная арифметика

В математике очень часто используются смешанные арифметические операции с целыми и вещественными числами: мы пишем A = 2pi*r, а не A = 2.0pi*r. При вычислении смешанные операции с целыми числами и числами с плавающей точкой должны выполняться с некоторой осторожностью. Предпочтительнее вторая форма, потому что 2.0 можно хранить непосредственно как константу с плавающей точкой, а литерал 2 нужно было бы преобразовать к представлению с плавающей точкой. Хотя обычно это делается компилято-ром автоматически, лучше точно написать, что именно вам нужно.

Другой потенциальный источник затруднений — различие между целочисленным делением и делением с плавающей точкой:

```
I:  Integer := 7;
J:  Integer := I / 2;
K: Integer := lnteger(Float(l) / 2.0);
```

Ada

Выражение в присваивании J задает целочисленное деление; результат, ко-нечно, равен 3. В присваивании К требуется деление с плавающей точкой: ре-зультат равен 3.5, и он преобразуется в целое число путем округления до 4.

В языках даже нет соглашений относительно того, как преобразовывать значения с плавающей точкой в целочисленные. Тот же самый пример на языке С выглядит так:

```
int i = 7;
int j = i/2;
int k = (int) ((float i)/ 2.0);
```

C

Здесь 3 присваивается как j, так и k, потому что значение 3.5 с плавающей точкой обрезается, а не округляется!

В языке С неявно выполняется смешанная арифметика, в случае необходимости целочисленные типы преобразуются к типам с плавающей точкой, а более низкая точность к более высокой. Кроме того, значения неявно преобразуются при присваивании. Таким образом, вышеупомянутый пример можно было бы написать как

```
int k = i/2.0;
```
C

«Продвижение» целочисленного i к плавающему типу вполне распознаваемо, и тем не менее для лучшей читаемости программ в присваиваниях (в отличие от инициализаций) преобразования типов лучше задавать явно:

```
k=(int)i/2.0;
```
C

В Ada *вся* смешанная арифметика запрещена; однако любое значение числового типа может быть явно преобразовано в значение любого другого числового типа, как показано выше.

Если важна эффективность, реорганизуйте смешанное выражение так, чтобы вычисление оставалось по возможности простым как можно дольше. Рассмотрим пример (вспомнив, что литералы в С рассматриваются как double):

```
int i,j,k,l; float f= 2.2 * i * j * k * I;
```
C

Здесь было бы выполнено преобразование i к типу double, затем умножение 2.2 * i и так далее для каждого целого числа, преобразуемого к типу double. Наконец, результат был бы преобразован к типу float. Эффективнее было бы написать:

```
int i j, k, I; I
float f=2.2F*(i*J*k*l);
```
C

чтобы гарантировать, что сначала будут перемножены целочисленные переменные с помощью быстрых целочисленных команд и что литерал будет храниться как float, а не как double. Конечно, такая оптимизация может привести к целочисленному переполнению, которого могло бы не быть, если вычисление выполнять с двойной точностью.

Одним из способов увеличения эффективности любого вычисления с плавающей точкой является изменение алгоритма таким образом, чтобы только часть вычислений должна была выполняться с двойной точностью. Например, физическая задача может использовать одинарную точность при вычислении движения двух объектов, которые находятся близко друг от друга (так что расстояние между ними можно точно представить относительно небольшим количеством цифр); программа затем может переключиться на двойную точность, когда объекты удалятся друг от друга.

9.3. Три смертных греха

Младший значащий разряд результата каждой операции с плавающей точкой может быть неправильным из-за ошибок округления. Программисты, кото-ре пишут программное обеспечение для численных расчетов, должны хоро-шо разбираться в методах оценки и контроля этих ошибок. Вот три грубые ошибки, которые могут произойти:

- исчезновение операнда,
- умножение ошибки,
- потеря значимости.

Операнд сложения или вычитания может исчезнуть, если он относительно мал по сравнению с другим операндом. При десятичной арифметике с пятью цифрами:

$$0.1234 \times 10^3 + 0.1234 \times 10^{-4} = 0.1234 \times 10^3$$

Маловероятно, что преподаватель средней школы учил вас, что x + y = x для ненулевого y, но именно это здесь и произошло!

Умножение ошибки — это большая абсолютная ошибка, которая может появиться при использовании арифметики с плавающей точкой, даже если относительная ошибка мала. Обычно это является результатом умножения деления. Рассмотрим вычисление *x • x:*

$$0.1234 \times 10^3 \bullet 0.1234 \times 10^3 = 0.1522 \times 10^5$$

и предположим теперь, что при вычислении *x* произошла ошибка на единицу младшего разряда, что соответствует абсолютной ошибке 0.1:

$$0.1235 \times 10^3 \bullet 0.1235 \times 10^3 = 0.1525 \times 10^5$$

Абсолютная ошибка теперь равна 30, что в 300 раз превышает ошибку перед умножением.

Наиболее грубая ошибка — полная потеря значимости, вызванная вычитанием почти равных чисел:

C

```
float f1= 0.12342;
float f2 = 0.12346;
```

В математике f2 -f1 = 0.00004, что, конечно, вполне представимо как четырехразрядное число с плавающей точкой: 0.4000×10^{-4}. Однако программа, вы-числяющая f2 - f 1 в четырехразрядном представлении с плавающей точкой, даст ответ:

$$0.1235\ 10° - 0.1234 \times 10° = 0.1000 \times 10^{-3}$$

что даже приблизительно не является приемлемым ответом.

Потеря значимости встречается намного чаще, чем можно было бы предположить, потому что проверка на равенство обычно реализуется вычитанием и последующим сравнением с нулем. Следующий условный оператор, таким образом, совершенно недопустим:

C

```
f2=...;
f2=…;
if (f1 ==f2)...
```

Самая невинная перестройка выражений для f 1 и f2, независимо от того, сделана она программистом или оптимизатором, может вызвать переход в условном операторе по другой ветке. Правильный способ проверки равенства с плавающей точкой состоит в том, чтобы ввести малую величину:

C

```
#define Epsilon10e-20
if ((fabs(f2-f1))<Epsilon)...
```

и затем сравнить абсолютное значение разности с малой величиной. По той же самой причине нет существенного различия между < = и < при вычислениях с плавающей точкой.

Ошибки в вычислениях с плавающей точкой часто можно уменьшить изменением порядка действий. Поскольку сложение производится слева направо, четырехразрядное десятичное вычисление

$$1234.0 + 0.5678 + 0.5678 = 1234.0$$

лучше делать как:

$$0.5678 + 0.5678 + 1234.0 = 1235.0$$

чтобы не было исчезновения слагаемых.

В качестве другого примера рассмотрим арифметическое тождество:

$$(x+y)(x-y)=x^2-y^2$$

и используем его для улучшения точности вычисления:

```
X, Y: Float_4;
Z: Float_7;
Z := Float_7((X + Y)*(X - Y));          -- Так считать?
Z := Float_7(X*X - Y*Y);                -- или так?
```

Ada

Если мы положим x = 1234.0 и y = 0.6, правильное значение этого выражения будет равно 1522755.64. Результаты, вычисленные с точностью до восьми цифр, таковы:

(1234.0 + 0.6) • (1234.0-0.6) =1235.0 • 1233.0=1522755.0

и

(1234.0 • 1234.0)-(0.6 • 0.6) = 1522756.0-0.36 =1522756.0

При вычислении $(x + y)(x- y)$ небольшая ошибка, являющаяся результатом сложения и вычитания, значительно возрастает при умножении. При вычислении по формуле $x^2 - y^2$ уменьшается ошибка от исчезновения слагаемого и результат получается более точным.

9.4. Вещественные типы в языке Ada

Замечание: техническое определение вещественных типов было значительно упрощено при переходе от Ada 83 к Ada 95, поэтому, если вы предполагаете детально изучать эту тему, лучше опускать более старые определения.

Типы с плавающей точкой в Ada

В разделе 4.6 мы описали, как можно объявить целочисленный тип, чтобы получить данный диапазон, в то время как реализация выбирается компилятором:

```
type Altitude is range 0 .. 60000;
```

Аналогичная поддержка переносимости вычислений с плавающей точкой обеспечивается объявлением произвольных типов с плавающей точкой:

```
type F is digits 12;
```

Это объявление запрашивает точность представления из 12 (десятичных) цифр. На 32-разрядном компьютере для этого потребуется двойная точность, тогда как на 64-разрядном компьютере достаточно одинарной точности. Об- ратите внимание, что, как и в случае целочисленных типов, это объявление создает новый тип, который нельзя использовать в операциях с другими типа-ми без явных преобразований.

Стандарт Ada подробно описывает соответствующие реализации такого Объявления. Программы, правильность которых зависит только от требо-ваний стандарта, а не от каких-либо причуд частной реализации, гаран-тированно легко переносимы с одного компилятора Ada на другой, даже на [компилятор для совершенно другой архитектуры вычислительной сис-темы.

Типы с фиксированной точкой в Ada

Тип с фиксированной точкой объявляется следующим образом:

type F is delta 0.1 range 0.0 .. 1.0;

Кроме диапазона при записи объявления типа с фиксированной точкой ука-зывается требуемая абсолютная погрешность в виде дроби после ключевого слова delta.

Заданные delta D и range R означают, что реализация должна предоставить набор *модельных чисел*, отличающихся друга от друга не больше чем на D и покрывающих диапазон R. На двоичном компьютере модельные числа были бы кратными ближайшего числа, меньшего D и являющегося степенью двойки, в нашем случае 1/16 = 0.0625. Данному выше объявлению соответствуют следующие модельные числа:

O, 1/16, 2/16,..., 14/16,15/16

Обратите внимание, что, даже если 1.0 определена как часть диапазона, это число не является одним из модельных чисел! Определение только требует, чтобы 1.0 лежала не далее 0.1 от модельного числа, и это требование выполняется, потому что 15/16 = 0.9375 и 1.0 — 0.9375 < 0.1.

Существует встроенный тип Duration, который используется для измерения временных интервалов. Здесь подходит тип с фиксированной точкой, потому что время будет иметь абсолютную погрешность (скажем 0.0001 с) в зависимости от аппаратных средств компьютера.

Для обработки коммерческих данных в Ada 95 определены *десятичные типы с фиксированной точкой.*

type Cost is delta 0.01 digits 10;

В отличие от обычных типов с фиксированной точкой, которые представляются степенями двойки, эти числа представляются степенями десяти и, таким образом, подходят для точной десятичной арифметики. Тип, объявленный выше, может поддерживать значения до 99999999.99.

9.5. Упражнения

1. Какие типы с плавающей точкой существуют на вашем компьютере? Перечислите диапазон и точность представления для каждого типа. Используется ли смещение в представлении экспоненты? Выполняется ли нормализация? Есть ли скрытый старший бит? Существует ли представление бесконечности или других необычных значений?

2. Напишите программу, которая берет число с плавающей точкой и печатает знак, мантиссу и экспоненту (после удаления всех смещений).

3. Напишите программу для целочисленного сложения и умножения с неограниченной точностью.

4. Напишите программу для печати двоичного представления десятичной дроби.

5. Напишите программу для BCD-арифметики.

6. Напишите программу для эмуляции сложения и умножения с плавающей точкой.

7. Объявите различные типы с фиксированной точкой в Ada и проверьте, как представляются значения. Как представляется тип Duration?

8. В Ada существуют ограничения на арифметику с фиксированной точкой. Перечислите и обоснуйте каждое ограничение.

Глава 10

Полиморфизм

Полиморфизм означает «многоформенность». Здесь мы этим термином обозначаем возможность для программиста использовать переменную, значение или подпрограмму двумя или несколькими различными способами. Полиморфизм почти по определению является источником ошибок; достаточно трудно понять программу даже тогда, когда каждое имя имеет одно значение, и намного труднее, если имя может иметь множество значений! Однако во многих случаях полиморфизм необходим и достаточно надежен при аккуратном применении.

Полиморфизм может быть статическим или динамическим. В статическом полиморфизме множественные формы разрешаются (конкретизируются) на этапе компиляции, и генерируется соответствующий машинный код. Например:

• преобразование типов: значение преобразуется из одного типа в другой;

• перегрузка (overloading): одно и то же имя используется для двух или нескольких разных объектов или подпрограмм (включая операции);

• родовой (настраиваемый) сегмент: параметризованный шаблон подпрограммы используется для создания различных конкретных экземпляров подпрограммы.

В динамическом полиморфизме структурная неопределенность остается до этапа выполнения:

• вариантные и неограниченные записи: одна переменная может иметь значения разных типов;

• диспетчеризация во время выполнения: выбор подпрограммы, которую нужно вызвать, делается при выполнении.

10.1. Преобразование типов

Преобразование типов — это операция преобразования значения одного типа к значению другого типа. Существуют два варианта преобразования типов: 1) перевод значения одного типа к допустимому значению другого типа, и 2) пересылка значения как неинтерпретируемой строки битов.

Преобразование числовых значений, скажем, значений с плавающей точкой, к целочисленным включает выполнение команд преобразования битов значения с плавающей точкой так, чтобы они представили соответствующее целое число. Фактически, преобразование типов делается функцией, получающей параметр одного типа и возвращающей результат другого типа. Синтаксис языка Ada для преобразования типов такой же, как у функции:

Ada

```
I: Integer := 5; F:
Float := Float(l);
```

в то время как синтаксис языка C может показаться странным, особенно в сложном выражении:

C

```
int i = 5;
float f = (float) i;
```

В C++ для совместимости сохранен синтаксис C, но для улучшения читаемо- сти программы также введен и функциональный синтаксис, как в Ada. Кроме того, и C, и C++ включают неявные преобразования между типами, прежде всего числовыми:

C

```
int i; float f = i;
```

Явные преобразования типов безопасны, потому что они являются всего лишь функциями: если не существует встроенное преобразование типа, вы всегда можете написать свое собственное. Неявные преобразования типов более проблематичны, потому что читатель программы никогда не знает, было преобразование преднамеренным или это просто оплошность. Использование целочисленных значений в сложном выражении с плавающей точкой не должно вызывать никаких проблем, но другие преобразования следует указывать явно.

Вторая форма преобразования типов просто разрешает программе исполь-зовать одну и ту же строку битов двумя разными способами. К сожалению, в языке C используется один и тот же синтаксис для обеих форм преобразова-ния: если преобразование типов имеет смысл, например между числовыми типами или указательными типами, то оно выполняется; иначе строка битов передается, как есть.

В языке Ada можно между любыми двумя типами осуществить *не контролируемое преобразование (unchecked conversion)*, при котором значение трактуется как неинтерпретируемая строка битов. Поскольку это небезопасно по самой сути и разрушает все с таким трудом добытые преимущества контроля типов, неконтролируемые преобразования не поощряются, и синтаксис языка спроектирован так, чтобы такие преобразования бросались в глаза. При просмотре программы вы не пропустите места таких преобразований и должны будете «оправдаться» хотя бы перед собой.

Хотя для совместимости в C++ сохранено такое же преобразование типов, как в C, в нем определен новый набор операций преобразования типов:

• dynamic_cast. См. раздел 15.3.

• static_cast. Выражение типа T1 может статически приводиться к типу T2, если T1 может быть неявно преобразовано к T2 или обратно; static_cast следует использовать для безопасных преобразований типов, как, например, float к int или обратно.

• reinterpret_cast. Небезопасные преобразования типов.

• const_cast. Используется, чтобы разрешить делать присваивания константным объектам.

10.2. Перегрузка

Перегрузка — это использование одного и того же имени для обозначения разных объектов в общей области действия. Использование одного и того же имени для переменных в двух разных процедурах (областях действия) не рассматривается как перегрузка, потому что две переменные не существуют одновременно. Идея перегрузки исходит из потребности использовать математические библиотеки и библиотеки ввода-вывода для переменных различных типов. В языке C имя функции вычисления абсолютного значения свое для каждого типа.

```
int   i  =abs(25);
double d=fabs( 1.57);
long   I  =labs(-25L);
```

C

В Ada и в C++ одно и то же имя может быть у двух или нескольких разных подпрограмм при условии, что *сигнатуры параметров* разные. Пока число и/или типы (а не только имена или режимы) формальных параметров различны, компилятор будет в состоянии запрограммировать вызов правильной подпрограммы, проверяя число и типы фактических параметров:

```
function Sin(X: in Float) return Float;
function Sin(X: in Long_Float) return Long_Float;

F1,F2: Float;
L1.L2: Long_Float:

F1 :=Sin(F2);
L1 :=Sin(L2);
```

Ada

Интересное различие между двумя языками состоит в том, что Ada прини-мает во внимание тип результата функции, в то время как C++ ограничивает-ся формальными параметрами:

```
|c++
float sin(float);
double sin(double);          // Перегрузка sin
double sin(float);           // Ошибка, переопределение в области действия
```

C++

Особый интерес представляет возможность перегрузки стандартных операций, таких как + и в Ada:

```
I Ada  function "+" (V1, V2: Vector) return Vector;
```

C++

Конечно, вы должны представить саму функцию, реализующую перегруженную операцию для новых типов. Обратите внимание, что синтаксические свойства операций, в частности старшинство, не изменяются. В C++ есть аналогичное средство перегрузки:

```
Vector operator + (const Vector &, const Vector &);
```

C++

Это совершенно аналогично объявлению функции, за исключением зарезервированного ключевого слова operator. Перегружать операции имеет смысл только в том случае, если вновь вводимые операции аналогичны предопределенным, иначе можно запутать тех, кто будет сопровождать программу.

При аккуратном использовании перегрузка позволяет уменьшить длины имен и обеспечить переносимость программы. Она может даже увеличить прозрачность программы, поскольку такие искусственные имена, как fabs, больше не нужны. С другой стороны, перегрузка без разбора может легко нарушить читаемость программы (если одному и тому же имени будет присваиваться слишком много значений). Перегрузка должна быть ограничена подпрограммами, выполняющими аналогичные вычисления, чтобы читатель программы мог понять смысл уже по самому имени подпрограммы.

10.3. Родовые (настраиваемые) сегменты

Массивы, списки и деревья — это структуры данных, в которых могут храниться элементы данных произвольного типа. Если нужно хранить несколько типов одновременно, необходима некоторая форма динамического полиморфизма. Однако если мы работаем только с гомогенными структурами данных, как, например, массив целых чисел или список чисел с плавающей точкой, достаточно статического полиморфизма, чтобы создавать экземпляры программ по шаблонам во времени компиляции.

Рассмотрим подпрограмму, сортирующую массив. Тип элемента массива используется только в двух местах: при сравнении и перестановке элементов.
Сложная обработка индексов делается одинаково для всех типов элементов массива:

```
type lnt_Array is array(lnteger range <>) of Integer;

procedure Sort(A: lnt_Array) is

      Temp, Min: Integer;
Begin                                                    Ada

for I in A'First ..A'Last-1 loop
     Min:=l;
     for J in I+1 .. A'Last loop
        if A(J) < A(Min) then Min := J; end if;
                                        -- Сравнить элементы, используя "<"
      end loop;
      Temp := A(l); A(l) := A(Min); A(Min) := Temp;
                                       -- Переставить элементы, используя ":="
end loop;
end Sort;
```

На самом деле даже тип индекса не существенен при программировании этой процедуры, лишь бы он был дискретным типом (например, символьным или целым).

Чтобы получить процедуру Sort для некоторого другого типа элемента, например Character, можно было бы физически скопировать код и сделать необходимые изменения, но это могло бы привести к дополнительным ошибкам. Более того, если бы мы хотели изменить алгоритм, то пришлось бы сделать эти изменения отдельно в каждой копии. В Ada определено средство, называемое *родовыми сегментами (generics)*, которое позволяет программисту задать шаблон подпрограммы, а затем создавать конкретные экземпляры подпрограммы для нескольких разных типов. Хотя в C нет подобного средства, его отсутствие не так серьезно, потому что указатели void, оператор sizeof и указатели на функции позволяют легко запрограммировать «обобщенные», пусть и не такие надежные, подпрограммы. Обратите внимание, что применение родовых сегментов не гарантирует, что конкретные экземпляры одной родовой подпрограммы будут иметь общий объектный код; фактически, при реализации может быть выбран независимый объектный код для каждого конкретного случая.

Ниже приведено объявление *родовой подпрограммы* с двумя *родовыми формальными параметрами:*

```
generic
   type Item is (<>);
   type ltem_Array is array(lnteger range <>) of Item;          Ada
procedure Sort(A: ltem_Array);
```

Это обобщенное объявление на самом деле объявляет не процедуру, а только шаблон процедуры. Необходимо обеспечить тело процедуры: оно будет написано в терминах родовых параметров:

Ada

```
procedure Sort(A: ltem_Array) is
    Temp, Min: Item;
begin
…                                   -- Полностью совпадает с вышеприведенным
end Sort;
```

Чтобы получить (подлежащую вызову) процедуру, необходимо *конкретизировать* родовое объявление, т. е. создать экземпляр, задав родовые фактические параметры:

Ada

```
type lnt_Array is array(lnteger range <>) of Integer;
type Char_Array is array(lnteger range <>) of Character;

procedure lnt_Sort(A: lnt_Array) is new Sort(lnteger, lnt_Array);
procedure Char_Sort(A: Char_Array) is new Sort(Character, Char_Array);
```

Это реальные объявления процедур; вместо тела процедуры после объявления следует ключевое слово is, и тем самым запрашивается новая копия обобщенного шаблона.

Родовые параметры — это параметры *этапа компиляции*, и используются они компилятором, чтобы сгенерировать правильный код для конкретного экземпляра. Параметры образуют *контракт* между кодом родовой процедуры и ее конкретизацией. Первый параметр Item объявлен с записью (<>). Это означает, что конкретизация программы обещает применить дискретный тип, такой как Integer или Character, а код обещает использовать только операции, допустимые на таких типах. Так как на дискретных типах определены операции отношения, процедура Sort уверена, что «<» допустима. Второй обобщенный параметр ltem_Array — это предложение контракта, которое говорит: какой бы тип ни был задан для первого параметра, второй параметр должен быть массивом элементов этого типа с целочисленным индексом.

Модель контракта работает в обе стороны. Попытка выполнить арифметическую операцию «+» на значениях типа Item в родовом теле процедуры является ошибкой компиляции, так как существуют такие дискретные типы, как Boolean, для которых арифметические операции не определены. И обратно,родовая процедура не может быть конкретизирована с элементом массива типа запись, потому что операция «<» для записей не определена.

Цель создания модели контракта заключается в том, чтобы позволить программистам многократно применять родовые модули и избавить их от необходимости знать, как реализовано родовое тело процедуры. Уж если родовое тело процедуры скомпилировано, конкретизация может завершиться неуспешно, только если фактические параметры не удовлетворяют контракту. Конкретизация *не может быть* причиной ошибки компиляции в теле процедуры.

Шаблоны в C++

В языке C++ обобщения реализованы с помощью специального средства — *шаблона (template):*

```
template <class ltem_Array> void Sort(ltem_Array parm)
{
…
}
```

Здесь нет необходимости в явной конкретизации: подпрограмма создается неявно, когда она используется:

```
typedef int l_Array[100];
typedef char C_Array[100];
l_Array a;
C_Array c;

Sort(a);                    // Конкретизировать для целочисленных массивов
Sort(c);                    // Конкретизировать для символьных массивов
```

Явная конкретизация — это оптимизация, задаваемая программистом по желанию; в противном случае, компилятор сам решает, какие конкретизации необходимо сделать. Шаблоны могут быть конкретизированы только по типам и значениям, или, в более общем случае, по классам (см. гл. 14).

Язык C++ не использует модель контракта, поэтому конкретизация может закончиться неуспешно, вызвав ошибку компиляции в определении шаблона. Это затрудняет производство и поставку шаблонов как самостоятельных компонентов программного обеспечения.

Родовые параметры-подпрограммы в языке Ada

В Ada допускается, чтобы родовые параметры были подпрограммами. Пример программы сортировки может быть написан так:

```
generic
    type Item is private;
    type ltem_Array is array(lnteger range <>) of Item;
    with function "<"(X, Y: in Item) return Boolean;
procedure Sort(A: ltem_Array);
```

Контракт теперь расширен тем, что для реализации операции «<» должна быть предоставлена булева функция. А поскольку операция сравнения обеспечена, Item больше не нужно ограничивать дискретными типами, для которых эта операция является встроенной. Ключевое слово private означает, что любой тип, на котором определено присваивание и сравнение на равенство, может применяться при реализации:

```
type Rec is record . .. end record;
type Rec_Array is array(lnteger range <>) of Rec;
function "<"(R1, R2: in Rec) return Boolean;

procedure Rec_Sort(A: Rec_Array) is new Sort(Rec, Rec_Array, "<");
```

Внутри подпрограммы Sort присваивание является обычным поразрядным присваиванием для записей, а когда нужно сравнить две записи, вызывается функция «<». Эта обеспеченная программистом функция решит, является ли одна запись меньше другой.

Модель контракта в языке Ada очень мощная: типы, константы, переменные, указатели, массивы, подпрограммы и пакеты (в Ada 95) могут использоваться как родовые параметры.

10.4. Вариантные записи

Вариантные записи используются, когда во время выполнения необходимо интерпретировать значение несколькими разными способами. Ниже перечислены распространенные примеры.

• Сообщения в системе связи и блоках параметров в вызовах операционной системы. Обычно первое поле записи является кодом, значение которого определяет количество и типы остальных полей в записи.

• Разнородные структуры данных, такие как дерево, которое может содержать узлы разных типов.

Чтобы решать проблемы такого рода, языки программирования представляют новый класс типов, называемый *вариантными записями*, которые имеют *альтернативные* списки полей. Такая переменная может первоначально содержать значение одного варианта, а позже ей может быть присвоено значение другого варианта с совершенно другим набором полей. Помимо альтернативных могут присутствовать поля, которые являются общими для всех записей этого типа; такие поля обычно содержат код, с помощью которого программа определяет, какой вариант используется на самом деле. Предположим, что мы хотим создать вариантную запись, поля которой могут быть или массивом, или записью:

```
typedef int Arr[10];
typedef struct {
    float      f1;
    int i1;
}Rec;
```
C

Давайте сначала определим тип, который кодирует вариант:

```
typedef enum {Record_Code, Array_Code} Codes; 23
```
C

Теперь с помощью типа union (объединение) в С можно создать вариантную запись, которая сама может быть вложена в структуру, включающую общее поле тега, характеризующего вариант:

```
typedef struct {
Codes code;                    /* Общее поле тега */
union {                        /* Объединение с альтернативными полями */
     Агг a;                   /* Вариант массива */
     Rec r;                  /* Вариант записи */
    } data;
} S_Type;

S_Type s;
```
C

С точки зрения синтаксиса это всего лишь обычная вложенность записей и массивов внутри других записей. Различие состоит в реализации: полю data выделяется объем памяти, достаточный для самого большого поля массива a или поля записи r (см. рис. 10.1). Поскольку выделяемая память рассчитана на самое большое возможное поле, вариантные записи могут быть чрезвычайно

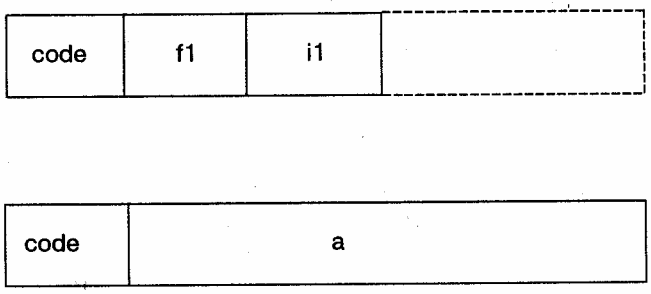

Рис. 10.1. Вариантные записи.

неэкономны по памяти, если один вариант очень большой, а другие маленькие:

```
union {
     int a[1000];
     float f;
      char c;
      }
```

C

Избежать этого можно ценой усложнения программирования — использовать указатель на длинные поля.

В основе вариантных записей лежит предположение, что в любой момент времени значимо только одно из полей объединения, в отличие от обычной записи, где все поля существуют одновременно:

```
if (s.code == Array_Code)
      i = s.data.a[4];              /* Выбор первого варианта */
else
       i = s.data.r.h ;             /* Выбор второго варианта */
```

C

Основная проблема с вариантными записями состоит в том, что они потенциально могут вызывать серьезные ошибки. Так как конструкция union позволяет программе обращаться к той же самой строке битов различными способами, то возможна обработка значения одного типа, как если бы это было значение какого-либо другого типа (скажем, обращение к числу с плавающей точкой, как к целому). Действительно, программисты, пишущие на языке Pascal, используют вариантные записи, чтобы делать преобразование типов, которое в языке непосредственно не поддерживается.

В вышеупомянутом примере ситуация еще хуже, потому что возможно обращение к ячейкам памяти, которые вообще не содержат никакого значения: поле s.data.r могло бы иметь длину 8 байт для размещения двух чисел, а поле s.data.a — 20 байт для размещения десяти целых чисел. Если в поле s.data.r в данный момент находится запись, то s.data.a[4] не имеет смысла.

В Ada не разрешено использовать вариантные записи, чтобы не разрушать контроль соответствия типов. Поле code, которое мы использовали в примере, теперь является обязательным полем, и называется *дискриминантом,* а при обращении к вариантным полям проверяется корректность значения дискриминанта. Дискриминант выполняет роль «параметра» типа:

```
type Codes is (Record_Code, Array_Code);

type S_Type(Code: Codes) is
 record
   case Code is
         when Record_Code => R: Rec;
         when Array_Code => A: Arr;
   end case;
 end record;
```
Ada

а запись должна быть объявлена с конкретным дискриминантом, чтобы компилятор точно знал, сколько памяти нужно выделить:

```
S1: S_Type(Record_Code);
S2: S_Type(Array_Code);
```
Ada

Другая возможность — объявить указатель на вариантную запись и проверять дискриминант во время выполнения:

I Ada type Ptr is access S_Type;

```
P: Ptr := new S_Type(Record_Code);
I:=P.R.I1;                          --Правильно
I:=P.A(5);                          -- Ошибка
```
Ada

Первый оператор присваивания правильный, поскольку дискриминант записи P.all — это Record_Code, который гарантирует, что поле R существует; в то же время второй оператор приводит к исключительной ситуации при работе программы, так как дискриминант не соответствует запрошенному полю.

Основное правило для дискриминантов в языке Ada заключается в том, что их можно читать, но не писать, так что нельзя обойти контроль соответствия типов. Это также означает, что память может выделяться в точном соответствии с выбранным вариантом, в отличие от обычного выделения для самого большого варианта.

Неограниченные записи в Ada

В дополнение к ограниченным записям, вариант которых при создании переменной фиксирован, Ada допускает объявление *неограниченных записей (unconstrained records)*, для которых допустимо во время выполнения безопасное с точки зрения контроля типов присваивание, хотя записи относятся к разным вариантам:

```
S1, S2: S_Type;                     -- Неограниченные записи
S1 := (Record_Code, 4.5);
S2 := (Array_Code, 1..10 => 17);
S1 := S2;                           -- Присваивание S1 другого варианта
                                    -- S2 больше, чем S1 !
```

Два правила гарантируют, что контроль соответствия типов продолжает работать:

• Для дискриминанта должно быть задано значение по умолчанию, чтобы гарантировать, что первоначально в записи есть осмысленный дискриминант:

```
type S_Type (Code: codes: = Record_Code) is ...
```

• Само по себе поле дискриминанта не может быть изменено. Допустимо только присваивание допустимого значения всей записи, как показано в примере.

Существуют две возможные реализации неограниченных записей. Можно создавать каждую переменную с размером максимального варианта, чтобы помещался любой вариант. Другая возможность — неявно использовать динамическую память из кучи. Если присваиваемое значение больше по размерам, то память освобождается и запрашивается большая порция. В большинстве реализаций выбран первый метод: он проще и не требует нежела- тельных в некоторых приложениях неявных обращений к менеджеру кучи.

10.5. Динамическая диспетчеризация

Предположим, что каждый вариант записи S_Type должен быть обработан собственной подпрограммой. Нужно использовать case-оператор, чтобы *пе-рейти (dispatch)* в соответствующую подпрограмму. Рассмотрим «диспетчерскую» процедуру:

```
procedure Dispatch(S: S_Type) is
begin
     case S.Code is                                          Ada
          when Record_Code => Process_Rec(S);
          when Array_Code => Process_Array(S);
     end case;
end Dispatch;
```

Предположим далее, что при изменении программы в запись необходимо до-бавить дополнительный вариант. Сделать изменения в программе нетрудно: бавить код к типу Codes, добавить вариант в case-оператор процедуры Dispatch и добавить новую подпрограмму обработки. Насколько легко сделать эти изменения, настолько они могут быть проблематичными в больших сис-темах, потому что требуют, чтобы *исходный* код существующих, хорошо про-веренных компонентов программы был изменен и перекомпилирован. Кроме того, вероятно, необходимо сделать повторное тестирование, чтобы гаранти-ровать, что изменение глобального типа перечисления не имеет непредусмот-ренных побочных эффектов.

Решением является размещение «диспетчерского» кода так, чтобы он был частью системы на этапе выполнения, поддерживающей язык, а не явно за-проограммированным кодом, как показано выше. Это называется динамиче-ским полиморфизмом, так как теперь можно вызвать общую программу Process(S), а привязку вызова конкретной подпрограммы отложить до этапа выполнения, когда станет известен текущий тег S. Этот полиморфизм под-держивается *виртуальными функциями (virtual functions)* в C++ и подпрограм-мами с *class-wide-параметрами* в Ada 95 (см. гл. 14).

10.6. Упражнения

1. Почему C++ не использует тип результата, чтобы различать перегруженные функции?

2. Какие задачи ставит перегрузка для компоновщика?

3. В C++ операции «++» и «--» могут быть как префиксными, так и постфиксными. Какова «подноготная» этой перегрузки, и как C++ справляется с этими операциями?

4. Ни Ada, ни C++ не позволяют с помощью перегрузки изменять старшинство или ассоциативность операций; почему?

5. Напишите шаблон программы сортировки на C++.

6. Напишите родовую программу сортировки на Ada и используйте ее для сортировки массива записей.

7. Первая родовая программа сортировки определила тип элемента (Item) как (О). Можно ли использовать Long_integer в конкретизации этой процедуры? А что можно сказать относительно Float?

8. Напишите программу, которая поддерживает разнородную очередь, то есть очередь, узлы которой могут содержать значения нескольких типов. Каждый узел будет вариантной записью с альтернативными полями для булевых, целочисленных и символьных значений.

Глава 11

Исключительные ситуации

11.1. Требования обработки исключительных ситуаций

Ошибка во время выполнения программы называется *исключительной ситуацией* или просто *исключением (exception).* Когда программы исполнялись не интерактивно (offline), соответствующая реакция на исключительную ситуацию состояла в том, чтобы просто напечатать сообщение об ошибке и завершить выполнение программы. Однако реакция на исключение в интерактивной среде не может быть ограничена сообщением, а должна также включать восстановление, например возврат к той точке, с которой пользователь может повторить вычисление или, по крайней мере, выбрать другой вариант. Программное обеспечение, которое используется в таких встроенных системах, как системы управления летательными аппаратами, должно выполнять восстановление при ошибках без вмешательства человека. Обработка исключений до недавнего времени, как правило, не поддерживалась в языках программирования; использовались только средства операционной системы. В этом разделе будут описаны некоторые механизмы обработки исключений, которые существуют в современных языках программирования.

Восстановление при ошибках не дается даром. Всегда есть затраты на дополнительные структуры данных и алгоритмы, необходимые для идентификации и обработки исключений. Кроме того, часто господствует точка зрения, что код обработки исключений сам является компонентом программы и может содержать ошибки, вызывающие более серьезные проблемы, чем первоначальное исключение! К тому же чрезвычайно трудно идентифицировать ситуации, приводящие к ошибке, и тестировать код обработки исключений, потому что сложно, а иногда и невозможно, создать ситуации, приводящие к ошибке.

Какие свойства делают средства обработки исключений хорошими?

• В случае отсутствия исключения издержки должны быть очень небольшими.

• Обработка исключения должна быть легкой в использовании и безопасной.

Первое требование важнее, чем это может показаться. Поскольку мы предполагаем, что исключительные ситуации, как правило, *не возникают*, издержки для прикладной программы должны быть минимальны. При наступлении исключительной ситуации издержки на ее обработку обычно не считаются существенными. Суть второго требования в том, что, поскольку исключения происходят нечасто, программирование реакции на них не должно требовать больших усилий; само собой разумеется, что обработчик исключения не должен использовать конструкции, которые могут привести к ошибке.

Одно предупреждение для программиста: обработчик исключений не является заменой условного оператора. Если ситуация может возникать, это *не является* ошибкой и должно быть явно запрограммировано. Например, вероятность того, что такие структуры данных, как список или дерево, окажутся пустыми, весьма велика, и эту ситуацию необходимо явно проверять, используя условный оператор:

if Ptr.Next= null then . . . else . . . Ada

С другой стороны, переполнение стека или потеря значимости в операциях с плавающей точкой встречается очень редко и почти наверняка указывает на ошибку в вычислениях.

В качестве элементарной обработки исключений в некоторых языках пользователю дана возможность определять блок кода, который будет выполнен перед завершением программы. Это полезно для наведения порядка (закрытия файлов и т.д.) перед выходом из программы. В языке С средство setjmp/longjmp позволяет пользователю задать дополнительные точки внутри программы, в которые обработчик исключений может возвращаться. Этого типа обработки исключений достаточно, чтобы гарантировать, что программа «изящно» завершится или перезапустится, но он недостаточно гибок для детализированной обработки исключений.

Обратите внимание, что, согласно нашему определению исключения как непредвиденной ошибки на этапе выполнения, в языке С «исключений» меньше, чем в таком языке, как Ada. Во-первых, такие ошибки, как выход за границы массива, не определены в языке С; они просто являются *ошибками программиста*, которые не могут быть «обработаны». Во-вторых, поскольку в С нет гибкого средства обработки исключений, каждая возможность языка, которая запрашивается через подпрограмму, возвращает код, указывающий, был запрос успешным или нет. Таким образом, в языке Ada распределение памяти с помощью new может вызвать исключительную ситуацию, если нет достаточного объема памяти, тогда как в С malloc возвращает код, который должен быть проверен явно. Выводы для стиля программирования следующие: в Ada можно использовать new обычным порядком, а обработку исключений проектировать независимо, в то время как в С полезно написать подпрограм-*му-оболочку* для malloc так, чтобы реакцию на исключительные ситуации можно было разработать и запрограммировать централизованно, вместо того чтобы разрешать каждому члену группы тестировать (или забывать тестировать) нехватку памяти:

```
void* get_memory(int n)
{                                                          C
void* p = malloc(n);
    if (p == 0)                          /* Выделение памяти потерпело неудачу */
                                         /* Сделайте что-нибудь или корректно
                                            завершите работу */
return p;
}
```

11.2. Исключения в PL/I

PL/1 был первым языком, который содержал средство для обработки исклю-чительных ситуаций в самом языке — *блок «при наступлении события»* или, |коротко, *«при» (on-unit).* Он является блоком кода, который выполняется, ког-да возникает исключительная ситуация; после его завершения вычисление продолжается. Проблема в PL/1, связанная с *блоком «при»,* состоит в том, что он влияет на обычные вычисления. Предположим, что активизирован блок, Срабатывающий при потере значимости с плавающей точкой. Тогда потенци-ально возможно воздействие на *каждое* выражение с плавающей точкой; дру-гими словами, каждое выражение с плавающей точкой содержит неявный вы-|зов блока и возврат из него. Это затрудняет выполнение оптимизации сохра-нения значений в регистрах или вынесения общих подвыражений.

11.3. Исключения в Ada

В языке Ada определен очень простой механизм обработки исключений, ко-
торый послужил моделью для других языков.
В Ada есть четыре предопределенных исключения:

Constraint_Error (ошибка ограничения). Нарушение ограничивающих ус-ловий, например, когда индексация массива выходит за границы или вы- бор вариантного поля не соответствует дискриминанту.

Storage_Error (ошибка памяти). Недостаточно памяти.

Program_Error (программная ошибка). Нарушение правил языка, напри-
мер выход из функции без выполнения оператора return.

Tasking_Error (ошибка задачи). Ошибки, возникающие при взаимодейст- вии задач
(см. гл. 12).

Конечно, Constraint_Error — наиболее часто встречающееся исключение, связанное со строгим контролем соответствия типов в языке Ada. Кроме того,
программист может объявлять исключения, которые обрабатываются точно
так же, как и предопределенные исключения.

Когда исключительная ситуация наступает, в терминологии языка Ada —*возбуждается (raised),* вызывается блок кода, называемый *обработчиком исключения (exeption handler).* В отличие от PL/I вызов обработчика *завершает* включающую процедуру. Так как обработчик не возвращается к нормальному вычислению, нет никаких помех для оптимизации. В отличие от обработчиков глобальных ошибок в C, обработка исключительных ситуаций в Ada чрезвычайно гибкая, потому что обработчики исключений могут быть привязаны к любой подпрограмме:

```
procedure Main is
    procedure Proc is
        P: Node_Ptr;
    begin
            P := new Node;              -- Может возбуждаться исключение
            Statement_1;                -- Пропускается, если возбуждено исключение
    exception
            when Storage_Error =>
-- Обработчик исключения
end Proc; begin Proc; Statement_2; — Пропускается, если исключение распространилось
из Proc
exception
when Storage_Error =>
-- Обработчик исключения
end Main;
```

После последнего исполняемого оператора подпрограммы ключевое слово exception вводит последовательность обработчиков исключений — по одному для каждого вида исключений. Когда возбуждается исключение, процедура покидается, и вместо нее выполняется обработчик исключения. Когда обработчик завершает свою работу, выполняется *нормальное* завершение процедуры. В приведенном примере программа выделения памяти может по-
родить исключительную ситуацию Storage_Error, в этом случае Statement_1 пропускается, и выполняется обработчик исключения. После завершения обработчика и *нормального* завершения процедуры главная программа продолжается с оператора Statement_2.

Семантика обработки исключений предоставляет программисту большую гибкость в управлении обработкой исключительных ситуаций:

• Если исключительная ситуация не обработана внутри процедуры, попытка ее выполнения оставляется, и исключительная ситуация возбуждается снова в точке вызова. При отсутствии обработчика в Proc исключение повторно было бы возбуждено в Main, оператор Statement_2 был бы пропущен и выполнен обработчик в Main.

• Если исключительная ситуация возбуждается *во время* выполнения обработчика, обработчик оставляется, и исключение возбуждается снова в точке вызова.

• У программиста есть выбор: возбудить то же самое или другое исключение в точке вызова. Например, мы могли бы перевести предопределенное исключение типа Storage_Error в исключение, определенное в прикладной программе. Это делается с помощью оператора rais в обработчике:

```
exception
    when Storage_Error =>
    …                          -- Обрабатывается исключение, затем
    raise Overflow;             --Возбуждается исключение Overflow в точке вызова
```

Обработчик для others может использоваться, чтобы обработать все исключения, которые не упомянуты в предыдущих обработчиках.

Если даже в главной программе нет обработчика для исключения, оно обрабатывается системой поддержки выполнения, которая обычно прерывает выполнение программы и выдает сообщение. Хорошим стилем программирования можно считать такой, при котором все исключения гарантированно обрабатываются хотя бы на уровне главной программы.

Определение исключений в языке Ada 83 не позволяло обработчику иметь доступ к информации о ситуации. Если более одной исключительной ситуации обрабатываются одинаково, никаким способом нельзя было узнать, что же именно произошло:

```
exception
   when Ex_1 | Ex_2 | Ex_3 =>
                          --Какое именно исключение произошло?
```

Язык Ada 95 позволяет обработчику исключительной ситуации иметь параметр:

```
exception
     when Ex: others =>
```

Всякий раз при возбуждении исключения параметр Ex будет содержать информацию, идентифицирующую исключение, а предопределенные процедуры позволят программисту отыскать информацию относительно исключения. Эта информация также может быть определена программистом (см. справочное руководство по языку, раздел 11.4.1).

Реализация

Реализуются обработчики исключений очень эффективно. Процедура, которая содержит обработчики исключений, имеет дополнительное поле в записи активации с указателем на обработчики (см. рис. 11.1). Требуется только одна команда при вызове процедуры, чтобы установить это поле, вот и все издержки при отсутствии исключений.

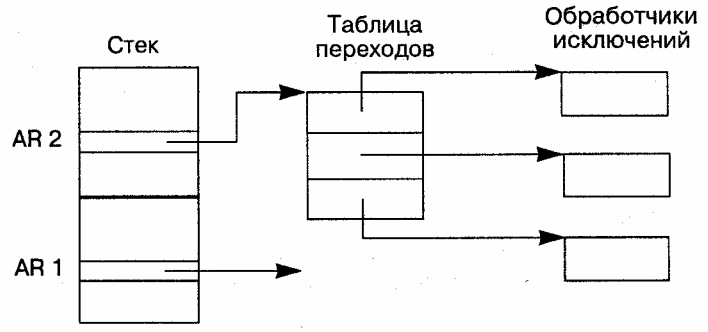

Рис. 11.1. Реализация обработчиков исключений.

Если исключение возбуждается, то, чтобы найти обработчик, может потребоваться большой объем вычислений для поиска по динамической цепочке, но, поскольку исключения происходят редко, это не представляет проблемы. Вспомните наш совет не использовать обработчик исключений как замену для гораздо более эффективного условного оператора.

11.4. Исключения в C++

Обработка исключений в C++ во многом сходна с той, которая применяется в языке Ada, а именно, исключение можно явно возбудить, обработать соответствующим обработчиком (если он есть), после чего блок (подпрограмма) окажется завершенным. Отличия в следующем:

• Вместо приписывания обработчика исключения к подпрограмме используется специальный синтаксис для указания группы операторов, к которым применяется обработчик.

• Исключения идентифицируются типом параметра, а не именем. Имеется специальный эквивалент синтаксиса others для обработки исключений, не упомянутых явно.

• Можно создавать семейства исключений, используя наследование (см. гл. 14).

• Если в языке Ada для исключения в программе не предусмотрен обработчик, то вызывается системный обработчик. В C++ программист может определить функцию terminate(), которая вызывается, когда исключение не обработано.

В следующем примере блок try идентифицирует область действия последовательности операторов, для которых обработчики исключений (обозначенные как catch-блоки) являются активными. Throw-оператор приводит к возбуждению исключений; в этом случае оно будет обработано вторым catch-блоком, так как строковый параметр throw-оператора соответствует параметру char* второго catch-блока:

```
void proc()
{
      …                                  // Исключения здесь не обрабатываются
      try {
       …
      throw "Invalid data";              // Возбудить исключение
     }
      catch (int i) {
      …                                  // Обработчик исключения
      }
       catch (char *s) {
      …                                  // Обработчик исключения
      }
       catch (...) {                     // Прочие исключения
      ….                                 // Обработчик исключений
     }
}
```

Как в Ada, так и в C++ допускается, чтобы обработчик вызывался для исключения, которое он не может видеть, потому что оно объявлено в теле пакета (Ada), или тип объявлен как private в классе (C++). Если исключение не обработано и в others (или ...), то оно будет неоднократно повторно возбуждаться до тех пор, пока, наконец, с ним не обойдутся как с необработанным исключением. В C++ есть способ предотвратить такую неопределенность поведения с помощью точного объявления в подпрограмме, какие исключительные ситуации она готова обрабатывать:

```
void proc() throw (t1 , t2, t3);
```

Такая *спецификация исключений (exception specifications)* означает, что отсутствующие в списке исключения, которые, возможно, будут возбуждаться, но не будут обрабатываться в рroc (или любой подпрограмме, вызванной из рroc), немедленно вызывут глобально определенную функцию unex-pectedQ вместо того, чтобы продолжать поиск обработчика. В больших системах эта конструкция полезна для документирования полного интерфейса подпрограмм, включая исключения, которые будут распространяться.

11.5. Обработка ошибок в языке Eiffei

Утверждения

В языке Eiffei подход к обработке исключений основан на концепции, что, прежде всего, ошибок быть не должно. Конечно, все программисты борются за это, и особенность языка Eiffei состоит в том, что в него включена поддержка определения правильности программы. Она основана на понятии *утверждений (assertions),* которые являются логическими формулами и обычно используются для формализации программы, но не являются непосредственно частью ее (см. раздел 2.2). Каждая подпрограмма, называемая *рутиной (routine)* в Eiffei, может иметь связанные с ней утверждения. Например, подпрограмма для вычисления результата *(result)* и остатка *(remainder)* целочисленного деления делимого *(dividend)* на делитель *(divisor)* была бы написана следующим образом:

```
integer_division(dividend, divisor, result, remainder: INTEGER) is
    require
       divisor > 0
  do
    from
        result = 0; remainder = dividend;
    invariant
        dividend = result*divisor + remainder
    variant
        remainder
    until
        remainder < divisor
    loop
        remainder := remainder - divisor;
        result := result + 1 ;
    end
  ensure
     dividend = result*divisor + remainder;
      remainder < divisor
end
```

Конструкция require (требуется) называется *предусловием* и специфицирует, какие выходные данные подпрограмма считает правильными. Конструкция do (выполнить) содержит выполняемые операторы, собственно и составляющие программу. Конструкция ensure (гарантируется) называется *постусловием* и содержит. условия, истинность которых подпрограмма обещает обеспечить, если будет выполнена конструкция do над данными, удовлетворяющими предусловию. В данном случае справедливость постусловия является достаточно тривиальным следствием инварианта (см. 6.6) и условия until.

На большем масштабе вы можете присоединить инвариант к классу (см. раздел 15.4). Например, класс, реализующий стек с помощью массива, включал бы инвариант такого вида:

```
invariant
  top >= 0;
  top < max;
```

Все подпрограммы класса должны гарантировать, что инвариант истинен, когда объект класса создан, и что каждая подпрограмма сохраняет истиность инварианта. Например, подпрограмма pop имела бы предусловие top> 0, в противном случае выполнение оператора:

```
top := top - 1
```

могло бы нарушить инвариант.

Типы перечисления

Инварианты применяются также, чтобы гарантировать безопасность типа, которая достигается в других языках использованием типов перечисления. Следующие объявления в языке Ada:

```
type Heat is (Off, Low, Medium, High);
Dial: Heat;
```

Ada

были бы записаны на языке Eiffel как обычные целые переменные с именованными константами:

```
Dial:          Integer;
Off:           Integer is 0;
Low:           Integer is 1;
Medium:        Integer is 2;
High:          Integer is 3;
```

Инвариант гарантирует, что бессмысленные присваивания не выполнятся:

```
invariant
Off <= Dial <= High
```

Последняя версия языка Eiffel включает *уникальные константы (unigue constants),* похожие на имена перечисления в том отношении, что их фактические значения присваиваются компилятором. Однако они по-прежнему остаются целыми числами, поэтому безопасность типа должна по-прежнему обеспечиваться с помощью утверждений: постусловие должно присоединяться к любой подпрограмме, которая изменяет переменные, чьи значения должны быть ограничены этими константами.

Проектирование по контракту

Утверждения — базовая компонента того, что язык Eiffel называет *проектированием по контракту,* в том смысле, что проектировщик подпрограммы заключает неявный контракт с пользователем подпрограммы: если вы обеспечите состояние, которое удовлетворяет предусловию, то я обещаю преобразовать состояние так, чтобы оно удовлетворяло постусловию. Точно так же класс поддерживает истинность своих инвариантов. Если контракты используются в системе повсеместно, то ничто никогда не может идти неправильно.

На практике, конечно, разработчик может потерпеть неудачу, пытаясь создать соответствующую контракту подпрограмму (либо потому, что операторы не удовлетворяют утверждениям, либо потому, что были выбраны неправильные утверждения). Для отладки и тестирования в реализации языка Eiffel для пользователя предусмотрена возможность запросить проверку утверждений при входе в подпрограмму и выходе из нее так, чтобы можно было остановить выполнение, если утверждение неверно.

Исключения

Подпрограммы Eiffel могут содержать обработчики исключений:

```
proc is
   do
     …                          -- Может возбуждаться исключение
    rescue
     …                          -- Обработчик исключения
end;
```

Когда возбуждается исключение, считается, что подпрограмма потерпела неудачу, и выполняются операторы после rescue. В отличие от языка Ada, после завершения обработчика исключение порождается снова в вызывающей программе. Это эквивалентно завершению в Ada обработчика исключения raise-оператором, который повторно порождает

в вызывающей подпрограмме то же самое исключение, которое заставило вызвать обработчик.

Мотивировка такого решения в предположении, что постусловие подпрограммы (и/или инвариант класса) удовлетворяются.

Если это не так, то вам, возможно, захочется получить уведомление, но уж наверняка вы *не можете* удовлетворить постусловие, т. е. потерпели неудачу при выполнении работы, которую заказала вам вызывающая подпрограмма. Другими словами, если.вам известно, как справиться с проблемой и удовлетворить постусловие, то предусмотрите это в подпрограмме. Это аналогично нашему совету не пользоваться исключениями вместо операторов if.

Обработчик исключения для помощи в решении возникших проблем может вносить некоторые изменения и запрашивать повторное выполнение подпрограммы с самого начала, если в него включено ключевое слово retry в качестве последнего оператора. Новая попытка может привести или не привести к успеху. Принципиально здесь то, что успешное выполнение — это нормальное завершение подпрограммы с выполненным постусловием. В противном случае ее выполнение терпит неудачу.

Обработчик исключений в языке Ada можно смоделировать в Eiffel следующим образом, хотя это идет вразрез с философией языка:

```
proc is
    local
        tried: Boolean;                          -- Инициализировано как false;
    do
        if not tried then
            -- Обычная обработка
            -- Порождает исключения
        else
            -- «Обработчик исключения»
        end
    rescue
        if not tried then
            tried := true;                       -- Чтобы не было двойного повтора
            retry
        end
end;
```

11.6. Упражнения

1. Пакет языка Ada. Исключения в Ada 95 определяют типы и подпрограммы для сопоставления информации с исключениями. Сравните эти конструкции с конструкциями throw и catch в C++.

2. Покажите, что исключение в языке Ada может быть порождено вне области действия исключения. (Подсказка: см. гл. 13.) Как можно обработать исключение, объявление которого не находится в области действия?

3. Покажите, как описание исключений на языке C++: void proc() throw(t1, t2, t3); может быть смоделировано с помощью многократных catch-блоков.

4. Изучите класс EXCEPTION в языке Eiffel и сравните его с обработчиком исключения в языке Ada.

Глава 12

Параллелизм

12.1. Что такое параллелизм?

Компьютеры с несколькими центральными процессорами (ЦП) могут выполнять несколько программ или компонентов одной программы *параллельно*. Вычисление, таким образом, может завершиться за меньшее *время счета* (количество часов), чем на компьютере с одним ЦП, с учетом затрат дополнительного времени ЦП на синхронизацию и связь. Несколько программ могут также совместно использовать компьютер с одним ЦП, так быстро переключая ЦП с одной программы на другую, что возникает впечатление, будто они выполняются одновременно. Несмотря на то, что переключение ЦП не реализует настоящую параллельность, удобно разрабатывать программное обеспечение для этих систем так, как если бы выполнение программ действительно происходило параллельно. Параллелизм — это термин, используемый для обозначения одновременного выполнения нескольких программ без уточнения, является вычисление параллельным на самом деле или только таким кажется.

Прямой параллелизм знаком большинству программистов в следующих формах:

• *Мультипрограммные (multi-programming)* операционные системы дают возможность одновременно использовать компьютер нескольким пользователям. Системы *разделения времени*, реализованные на обычных больших машинах и миникомпьютерах, в течение многих лет были единственным способом сделать доступными вычислительные средства для таких больших коллективов, как университеты.

• *Многозадачные (multi-tasking)* операционные системы дают возможность одновременно выполнять несколько компонентов одной программы (или программ одного пользователя). С появлением персональных компьютеров мультипрограммные компьютеры стали менее распространенными, но даже одному человеку часто необходим многозадачный режим для одновременного выполнения разных задач, как, например, фоновая печать при диалоговом режиме работы с документом.

• *Встроенные системы (embedded systems)* на заводах, транспортных системах и в медицинской аппаратуре управляют наборами датчиков и приводов в «реальном масштабе времени». Для этих систем характерно требование, чтобы они выполняли относительно небольшие по объему вычисления через очень короткие промежутки времени: каждый датчик должен быть считан и проинтерпретирован, затем программа должна выбрать соответствующее действие, и, наконец, данные в определенном формате должны быть переданы к приводам. Для реализации встроенных систем используются многозадачные операционные системы, позволяющие координировать десятки обособленных вычислений.

Проектирование и программирование параллельных систем являются чрезвычайно сложным делом, и целые учебники посвящены различным аспектам этой проблемы: архитектуре систем, диспетчеризации задач, аппаратным интерфейсам и т. д. Цель этого раздела состоит в том, чтобы дать краткий обзор *языковой поддержки параллелизма*, который традиционно обеспечивался функциями операционной системы и аппаратурой.

Параллельная программа (concurent program) состоит из одного или нескольких программных компонентов (процессов), которые могут выполняться параллельно. Параллельные программы сталкиваются с двумя проблемами:

Синхронизация. Даже если процессы выполняются одновременно, иногда один процесс должен будет синхронизировать свое выполнение с другими процессами. Наиболее важная форма синхронизации — взаимное исключение: два процесса не должны обращаться к одному и тому же ресурсу (такому, как диск или общая таблица) одновременно.

Взаимодействие. Процессы не являются полностью независимыми; они должны обмениваться данными. Например, в программе управления полетом процесс, считывающий показания датчика высоты, должен передать результат процессу, который делает расчеты для автопилота.

12.2. Общая память

Самая простая модель параллельного программирования — это модель с *общей памятью* (см. рис. 12.1). Два или несколько процессов могут обращаться к одной и той же области памяти, хотя они также могут иметь свою собственную частную, или приватную, (private) память. Предположим, что у нас есть два процесса, которые пытаются изменить одну и ту же переменную в общей памяти:

```
procedure Main is
   N: Integer := 0;
   task T1;
   task T2;
```

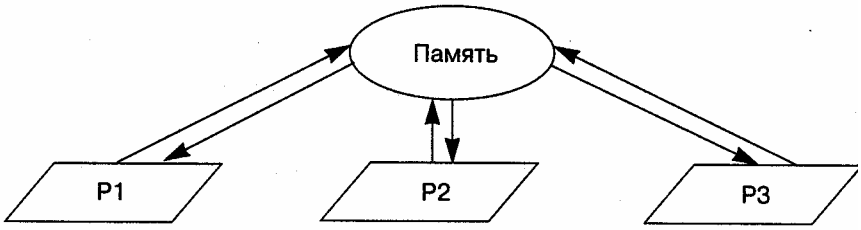

Рис. 12.1. Модель параллелизма с общей памятью.

```
task body T1 is
begin
     for I in 1 ..100 loop N := N+1; end loop;
end T1;

task body T2 is
begin
      for I in 1 ..100 loop N := N+1; end loop;
end T2;

begin
null;
end Main;
```

Рассмотрим теперь реализацию оператора присваивания:

```
load    R1,N    Загрузить из памяти
add     R1,#1   Увеличить содержимое регистра
store   R1,N    Сохранить в памяти
```

Если каждое выполнение тела цикла в T1 завершается до того, как T2 выполняет свое тело цикла, N будет увеличено 200 раз. Однако каждая задача может быть выполнена на отдельном компьютере со своим набором регистров. В этом случае может иметь место следующая последовательность событий:

• T1 загружает N в свой регистр R1 (значение равно и).

• T2 загружает N в свой регистр R1 (значение равно «).

• T1 увеличивает R1 (значение равно *n* + 1).

• T2 увеличивает R1 (значение равно и + 1).

• T1 сохраняет содержимое своего регистра R1 в N (значение равно *n* + 1).

• T2 сохраняет содержимое своего регистра R1 в N (значение равно *n* + 1).

Результат выполнения каждого из двух тел циклов состоит только в том, что N увеличится на единицу. Результирующее значение N может лежать между 100 и 200 в зависимости от относительной скорости каждого из двух процессоров.

Важно понять, что это может произойти даже на компьютере, который реализует многозадачный режим путем использования единственного ЦП. Когда ЦП переключается с одного процесса на другой, регистры, которые используются заблокированным процессом, сохраняются, а затем восстанавливаются, когда этот процесс продолжается.

В теории параллелизма выполнение параллельной программы определяется как *любое чередование атомарных команд* задач. Атомарная команда — это всего лишь команда, которую нельзя выполнить «частично» или прервать, чтобы продолжить выполнение другой задачи. В модели параллелизма с общей памятью команды загрузки и сохранения являются атомарными.

Если говорить о чередующихся вычислениях, то языки и системы, которые поддерживают параллелизм, различаются уровнем определенных в них атомарных команд. Реализация команды должна гарантировать, что она выполняется атомарно. В случае команд загрузки и сохранения это обеспечивается аппаратным интерфейсом памяти. Атомарность команд высокого уровня реализуется с помощью базисной системы поддержки времени выполнения и поддерживается специальными командами ЦП.

12.3. Проблема взаимных исключений

Проблема взаимных исключений (mutual exclusion problem) для параллельных программ является обобщением приведенного выше примера. Предполагается, что в каждой задаче (из параллельно выполняемых) вычисление делится на *критическую (critical)* и *некритическую (non-critical) секцию (section),* которые неоднократно выполняются:

```
task body T_i is
begin
    loop
      Prologue;
     Critical_Section;
     Epilogue;
     Non_Critical_Section;
    end loop;
end T_i:
```

Для решения проблемы взаимных исключений мы должны найти такие последовательности кода, называемые *прологом (prologue)* и *эпилогом (epilogue)*, чтобы программа удовлетворяла следующим требованиям, которые должны выполняться для всех чередований последовательностей команд из набора задач:

Взаимное исключение. В любой момент времени только одна задача выполняет свою критическую секцию.

Отсутствие взаимоблокировки (no deadlock). Всегда есть, по крайней мере, одна задача, которая в состоянии продолжить выполнение.

Жизнеспособность. Если задаче необходимо выполнить критическую секцию, в *конце концов* она это сделает.

Справедливость. Доступ к критическому участку предоставляется «по справедливости».

Существуют варианты решения проблемы взаимных исключений, использующие в качестве атомарных команд только load (загрузить) и store (сохранить), но эти решения трудны для понимания и выходят за рамки данной книги, поэтому мы отсылаем читателя к учебникам по параллельному программированию.

Э. Дейкстра (E.W. Dijkstra) определил абстракцию синхронизации высокого уровня, называемую *семафором*, которая тривиально решает эту проблему. Семафор S является переменной, которая имеет целочисленное значение; для семафоров определены две *атомарные* команды:

```
Wait(S):        when S > 0 do S := S -1;
Signal(S):      S:=S+1;
```

Процесс, выполняющий команду Wait(S), блокируется на время, пока значение S неположительно. Обратите внимание, что, поскольку команда является атомарной, то, как только процесс удостоверится, что S положительно, он сразу уменьшит S (до того, как любой другой процесс выполнит команду!). Точно так же Signal(S) выполняется атомарно без возможности прерывания другим процессом между загрузкой и сохранением S. Проблема взаимных исключений решается следующим образом:

Ada

```
procedure Main is
    S: Semaphore := 1 ;
    task T_i;                          -- Одна из многих
    task body T_i is
    begin
        loop
          Wait(S);
          Critical_Section;
          Signal(S);
          Non_Critical_Section;
       end loop;
   end T_i;
begin
    null;
end Main;
```

Мы предлагаем читателю самостоятельно убедиться в том, что это решение является правильным.

Конечно, самое простое — это переложить бремя решения проблемы на разработчика системы поддержки этапа выполнения.

12.4. Мониторы и защищенные переменные

Проблема, связанная с семафорами и аналогичными средствами, обеспечиваемыми операционной системой, состоит в том, что они не структурны. Если нарушено соответствие между Wait и Signal, программа может утратить синхронизацию или блокировку. Для решения проблемы структурности была разработана концепция так называемых *мониторов (monitors)*, и они реализованы в нескольких языках. Монитор — это совокупность данных и подпрограмм, которые обладают следующими свойствами:

• Данные доступны только подпрограммам монитора.

• В любой момент времени может выполняться не более одной подпрограммы монитора. Попытка процесса вызвать процедуру монитора в то время, как другой процесс уже выполняется в мониторе, приведет к приостановке нового процесса.

Поскольку вся синхронизация и связь выполняются в мониторе, потенциальные ошибки параллелизма определяются непосредственно программированием самого монитора; а процессы пользователя привести к дополнительным ошибкам не могут. Интерфейс монитора аналогичен интерфейсу операционной системы, в которой процесс вызывает монитор, чтобы запросить и получить обслуживание. Синхронизация процессов обеспечивается автоматически. Недостаток монитора в том, что он является централизованным средством.

Первоначально модель параллелизма в языке Ada (описанная ниже в разделе 12.7) была чрезвычайно сложной и требовала слишком больших затрат для решения простых проблем взаимных исключений. Чтобы это исправить, в Ada 95 были введены средства, аналогичные мониторам, которые называются *защищенными переменными (protected variables)*. Например, семафор можно смоделировать как защищенную переменную. Этот интерфейс определяет две операции, но целочисленное значение семафора рассматривает как *приватное* (private), что означает, что оно недоступно для пользователей семафора:

```
protected type Semaphore is
    entry Wait;
    procedure Signal;
private
    Value: Integer := 1;
end Semaphore;
```

Реализация семафора выглядит следующим образом:

```
protected body Semaphore is
    entry Wait when Value > 0 is
    begin
        Value := Value- 1;
     end Wait;
    procedure Signal is
    begin
       Value := Value + 1 ;
      end Signal;
end Semaphore;
```

Ada

Выполнение entry и procedure взаимно исключено: в любой момент времени только одна задача будет выполнять операцию с защищенной переменной. К тому же entry имеет *барьер (barrier),* который является булевым выражением. Задача, пытающаяся выполнить entry, будет заблокирована, если выражение имеет значение «ложь». Всякий раз при завершении защищенной операции все барьеры будут перевычисляться, и будет разрешено выполнение той задачи, барьер которой имеет значение «истина». В приведенном примере, когда Signal увеличит Value, барьер в Wait будет иметь значение «истина», и заблокированная задача сможет выполнить тело entry.

12.5. Передача сообщений

По мере того как компьютерные аппаратные средства дешевеют, *распределенное программирование* приобретает все большее значение. Программы разбиваются на параллельные компоненты, которые выполняются на разных компьютерах. Модель с разделяемой памятью уже не годится; проблема синхронизации и связи переносится на *синхронную передачу сообщений (synchronous message passing),* изображенную на рис. 12.2. В этой модели канал связи c может существовать между любыми двумя процессами. Когда один процесс посылает сообщение m в канал, он приостанавливается до тех пор, пока другой процесс не будет готов его получить. Симметрично, процесс, который ожидает получения сообщения, приостанавливается, пока посылающий процесс не готов послать. Эта приостановка используется для синхронизации процессов.

Синхронная модель параллелизма может быть реализована в самом языке программирования или в виде услуги операционной системы: потоки (pipes),

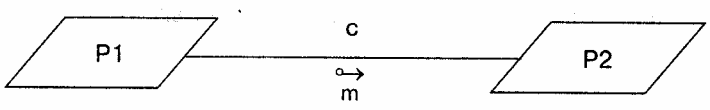

Рис. 12.2. Параллелизм с передачей сообщений (occam).

гнезда (sockets) и т.д. Модели отличаются способами, которыми процессы адресуют друг друга, и способом передачи сообщений. Далее мы опишем три языка, в которых методы реализации синхронного параллелизма существенно различны.

12.6. Язык параллельного программирования occam

Модель синхронных сообщений была первоначально разработана Хоаром (C. A. R. Hoare) в формализме, называющемся CSP (Communicating Sequential Processes — Взаимодействующие последовательные процессы). На практике он реализован в языке occat, который был разработан для программирования *транспьютеров* — аппаратной многопроцессорной архитектуры для распределенной обработки данных.

В языке occam адресация фиксирована, и передача сообщений односторонняя, как показано на рисунке 12.2. *Канал* имеет имя и может использоваться только для отправки сообщения из одного процесса и получения его в другом:

```
CHAN OF INT c :
PAR
  INT m:
  SEQ
                          -- Создается целочисленное значение m
  c! m
 INT v:
SEQ
   c? v
                           -- Используется целочисленное значение в v
```

c объявлено как канал, который может передавать целые числа. Канал должен использоваться именно в двух процессах: один процесс содержит команды вывода (c!), а другой — команды ввода (c?).

Интересен синтаксис языка occam. В других языках режим выполнения «по умолчанию» — это последовательное выполнение группы операторов, а для задания параллелизма требуются специальные указания. В языке occam параллельные и последовательные вычисления считаются в равной степени важными, поэтому вы должны явно указать, используя PAR и SEQ, как именно должна выполняться каждая группа (выровненных отступами) операторов.

Хотя каждый канал связывает ровно два процесса, язык occam допускает, чтобы процесс одновременно ждал передачи данных по любому из нескольких каналов:

```
[10]CHAN OF INT c :                    -- Массив каналов
ALT i = O FOR 10
     c[i] ? v
                                       -- Используется целочисленное значение в v
```

Этот процесс ждет передачи данных по любому из десяти каналов, а обработка полученного значения может зависеть от индекса канала.

Преимущество коммуникации точка-точка состоит в ее чрезвычайной эффективности, потому что вся адресная информация «скомпилирована». Не требуется никаких других средств поддержки во время выполнения кроме синхронизации процессов и передачи данных; в транспьютерных системах это делается аппаратными средствами. Конечно, эта эффективность достигается за счет уменьшения гибкости.

12.7. Рандеву в языке Ada

Задачи в языке Ada взаимодействуют друг с другом во время *рандеву (rendezvous).* Говорят, что одна задача T1 вызывает *вход (entry)* е в другой задаче T2 (см. рис. 12.3). Вызываемая задача должна выполнить accept-оператор для этого входа:

```
accept E(P1: in Integer; P2: out Integer) do
…
end E;
```

Когда задача выполняет вызов входа, и есть другая задача, которая уже выполнила accept для этого входа, имеет место рандеву.

• Вызывающая задача передает входные параметры принимающей задаче и затем блокируется.

• Принимающая задача выполняет операторы в теле accept.

• Принимающая задача возвращает выходные параметры вызывающей задаче.

• Вызывающая задача разблокируется.

Определение рандеву симметрично в том смысле, что, если задача выполняет accept-оператор, но ожидаемого вызова входа еще не произошло, она

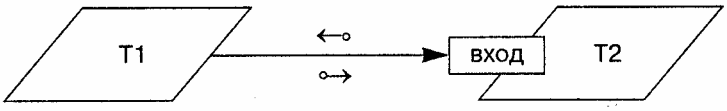

Рис. 12.3. Задачи и входы в Ada.

будет заблокирована, пока некоторая задача не вызывет вход для этого accept-оператора*.

Подчеркнем, что адресация осуществляется только в одном направлении: вызывающая задача должна знать имя принимающей задачи, но принимающая задача не знает имени вызывающей задачи. Возможность создания *серверов (servers),* т. е. процессов, предоставляющих определенные услуги любому другому процессу, послужила мотивом для выбора такого проектного решения. *Задача-клиент (client) должка,* конечно, знать название сервиса, который она запрашивает, в то время как задача-сервер предоставит сервис любой задаче, и ей не нужно ничего знать о клиенте.

Одно рандеву может включать передачу сообщений в двух направлениях, потому что типичный сервис может быть запросом элемента из структуры данных. Издержки на дополнительное взаимодействие, чтобы возвратить результат, были бы сверхмерными.

Механизм рандеву чрезвычайно сложен: задача может одновременно ждать вызова различных точек входа, используя select-оператор:

```
select
      accept El do ... end El;
or
      accept E2 do . . . end E2;
or
       accept E3 do . . . end E3;
end select;
```

Альтернативы выбора в select могут содержать булевы выражения, называемые *охраной (guards)*, которые дают возможность задаче контролировать, какие вызовы она хочет принимать. Можно задавать таймауты (предельные времена ожидания рандеву) и осуществлять опросы (для немедленной реакции в критических случаях). В отличие от конструкции ALT в языке occam, select-оператор языка Ada не может одновременно ожидать произвольного числа входов.

Обратите внимание на основное различие между защищенными переменными и рандеву:

• Защищенная переменная — это пассивный механизм, а его операции выполняются другими задачами.

• accept-оператор выполняется задачей, в которой он появляется, то есть он выполняет вычисление от имени других задач.

Рандеву можно использовать для программирования сервера и в том случае, если сервер делает значимую обработку *помимо* связи с клиентом:

```
task Server is
begin
    loop
        select
            accept Put(l: in Item) do
                -- Отправить I в структуру данных
            end Put;
         or
            accept Get(l: out Item) do
                -- Достать I из структуры данных
             end Get;
          end select;
                -- Обслуживание структуры данных
      end loop;
end Server;
```

Сервер отправляет элементы в структуру данных и достает их из нее, а после каждой операции он выполняет дополнительную обработку структуры данных, например регистрирует изменения. Нет необходимости блокировать другие задачи во время выполнения этой обработки, отнимающей много времени.

В языке Ada чрезвычайно гибкий механизм параллелизма, но эта гибкость достигается ценой менее эффективной связи, чем коммуникации точка-точка в языке occam. С другой стороны, в языке occam фактически невозможно реализовать гибкий серверный процесс, так как каждый дополнительный клиентский процесс нуждается в отдельном именованном канале, а это требует изменения программы сервера.

12.8. Linda

Linda — это не язык программирования как таковой, а модель параллелизма, которая может быть добавлена к существующему языку программирования. В отличие от однонаправленной (Ada) или двунаправленной адресации (occam), Linda вообще не использует никакой адресации между параллельными процессами! Вместо этого процесс может по выбору отправить сообщение в глобальную *кортежную область (Tuple Space).* Она названа так потому, что каждое сообщение представляет собой кортеж, т. е. последовательность из одного или нескольких значений, возможно, разных типов.

Например:

(True, 5.6, 'C', False)

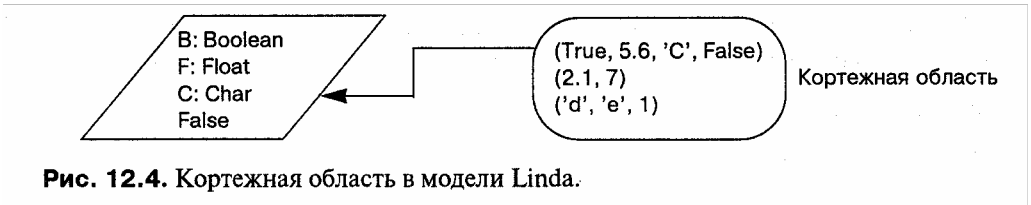

Рис. 12.4. Кортежная область в модели Linda.

— это четверной кортеж, состоящий из булева с плавающей точкой, символьного и снова булева значений.

Существуют три операции, которые обращаются к кортежной области:

out — поместить кортеж в кортежную область;

in — блокировка, пока не существует соответствующего кортежа, затем его удаление (см. рис. 12.4);

read — блокировка, пока не существует соответствующего кортежа (но без удаления его).

Синхронизация достигается благодаря тому, что команды in и read должны определять сигнатуру кортежа: число элементов и их типы. Только если кортеж существует с соответствующей сигнатурой, может быть выполнена операция получения, иначе процесс будет приостановлен. Кроме того, один или несколько элементов кортежа могут быть заданы явно. Если значение задано в сигнатуре, оно должно соответствовать значению в той же самой позиции кортежа; если задан тип, он может соответствовать любому значению этого типа в данной позиции. Например, все последующие операторы удалят первый кортеж в кортежной области на рис. 12.4:

```
in(True, 5.6, 'C', False)
in(B: Boolean, 5.6, 'C', False)
in(True, F: Float, 'C', Ё2: Boolean)
```

Второй оператор in возвратит значение True в формальном параметре B; третий оператор in возвратит значения 5.6 в F и False — в B2.

Кортежная область может использоваться для диспетчеризации вычислительных работ для процессов, которые могут находиться на разных компьютерах. Кортеж ("job", J, C) укажет, что работу J следует назначить компьютеру C. Каждый компьютер может быть заблокирован в ожидании работы:

```
in("job", J: Jobs, 4);                    -- Компьютер 4 ждет работу
```

Задача диспетчеризации может «бросать» работы в кортежную область. С помощью формального параметра оператора out можно указать, что безразлично, какой именно компьютер делает данную работу:

```
out("job", 6, C: Computers);              -- Работа 6 для любого компьютера
```

Преимущество модели Linda в чрезвычайной гибкости. Обратите внимание, что процесс может поместить кортеж в кортежную область и завершиться;

только позднее другой процесс найдет этот кортеж. Таким образом, Linda-программа распределена как во *времени*, так и в *пространстве* (среди процес-сов, которые могут быть на отдельных ЦП). Сравните это с языками Ada и occam, которые требуют, чтобы процессы непосредственно связывались друг с другом. Недостаток модели Linda состоит в дополнительных затратах на поддержку кортежной области, которая требует потенциально неограниченной глобальной памяти. Хотя кортежная область и является глобальной, бы-ли разработаны сложные алгоритмы для ее распределения среди многих процессоров.

12.9. Упражнения

1. Изучите следующую попытку решать проблему взаимного исключения в рамках модели с разделяемой памятью, где B1 и B2 — глобальные булевы переменные с начальным значением «ложь»:

Ada

```
task body T1 is
begin
    loop
        B1 :=True;
        loop
             exit when not B2;
             B1 := False;
             B1 :=True;
        end loop;
        Critical_Section;
        B1 := False;
       Non_Critical_Section;
     end loop;
end T1;

task body T2 is
begin
   loop
       B2 := True;
       loop
           exit when not B1;
           B2 := False;
           B2 := True;
       end loop;
       Critical_Section;
       B2 := False:
       Non_Critical_Section;
    end loop;
end T2;
```

Каков смысл переменных B1 и B2? Могут ли обе задачи находиться в своих критических областях в какой-нибудь момент времени? Может ли программа блокироваться? Достигнута ли жизнеспособность?

2. Проверьте решение проблемы взаимного исключения с помощью семафора. Покажите, что во *всех* чередованиях команд в любой момент времени в критической области может находиться не более одной задачи. Что можно сказать относительно взаимоблокировки, жизнеспособности и справедливости?

3. Что произойдет с решением проблемы взаимного исключения, если семафору задать начальное значение больше 1?

4. Попробуйте точно определить справедливость. Какая связь между справедливостью и приоритетом?

5. Как бы вы реализовали семафор?

6. Как диспетчер работ Linda обеспечивает, чтобы конкретная работа попадала на конкретный компьютер?

7. Напишите Linda-программу для умножения матриц. Получение каждого векторного произведения считайте отдельной «работой»; начальный процесс диспетчеризации заполняет кортежную область «работами»; рабочие процессы удаляют «работы» и возвращают результаты; заключительный процесс сбора удаляет и выводит результаты.

8. Переведите Linda-программу умножения матриц на язык Ada. Решите проблему дважды: один раз с отдельными задачами для диспетчера и сборщика и один раз в рамках единой задачи, которая выполняет обе функции в одном select-операторе.

4 Программирование больших систем

Глава 13

Декомпозиция программ

И начинающие программисты, и руководители проектов, экстраполируя ту простоту и легкость, с какой один человек может написать отдельную программу, часто полагают, что разработать программную систему также просто. Нужно только подобрать группу программистов и поручить им работу. Однако существует глубокая пропасть между написанием (небольших) программ и созданием (больших) программных систем, и многие системы поставляются с опозданием, с большим количеством ошибок и обходятся в несколько раз дороже, чем по первоначальной оценке.

Разработка программного обеспечения (software engineering) имеет дело с методами организации и управления группами разработчиков, с системой обозначений и инструментальными средствами, которые поддерживают этапы процесса разработки помимо программирования. Они включают этапы технического задания, проектирования и тестирования программного обеспечения.

В этой и двух последующих главах мы изучим конструкции языков программирования, которые разработаны для того, чтобы поддерживать создание больших программных систем. Не вызывает сомнений компромисс: чем меньшую поддержку предлагает язык для разработки больших систем, тем больше потребность в методах, соглашениях и системах обозначений, которые являются внешними по отношению к самому языку. Так как язык программирования, несомненно, необходим, кажется разумным включить поддержку больших систем в состав самого языка и ожидать, что компилятор по возможности автоматизирует максимальную часть процесса разработки. Мы, разработчики программного обеспечения, всегда хотим автоматизировать чью-нибудь чужую работу, но часто приходим в состояние неуверенности перед тем, как включить автоматизацию в языки программирования.

Главная проблема состоит в том, как разложить большую программную систему на легко управляемые компоненты, которые можно разработать отдельно и собрать в систему, где все компоненты взаимодействовали бы друг с другом, как запланировано. Начнем обсуждение с элементарных «механических» методов декомпозиции программы и перейдем к таким современным понятиям, как абстрактные типы данных и объектно-ориентированное программирование, которые направляют проектировщика системы на создание семантически значимых компонентов.

Перед тем как начать обсуждение, сделаем замечание для читателей, которые только начинают изучать программирование. Понятия будут продемонстрированы на небольших примерах, которые может вместить учебник, и вам может показаться, что это всего лишь излишняя «бюрократия». Будьте уверены, что поколениями программистов был пройден тяжелый путь, доказывающий, что такая бюрократия необходима; разница только в одном, либо она определена и реализована внутри стандарта языка, либо изобретается и внедряется администрацией для каждого нового проекта.

13.1. Раздельная компиляция

Первоначально декомпозиция программ делалась исключительно для того, чтобы дать возможность программисту раздельно компилировать компоненты программы. Благодаря мощности современных компьютеров и эффективности компиляторов эта причина теперь не столь существенна, как раньше, но важно изучить раздельную компиляцию, потому что для ее поддержки часто используются те же самые возможности, что и для декомпозиции программы на логические компоненты. Даже в очень больших системах, которые нельзя создать без раздельной компиляции, декомпозиция на компоненты делается при проектировании программы и не имеет отношения к этапу компиляции. Поскольку программные компоненты обычно относительно невелики, лимитирующим фактором при внесении изменений в программы обычно оказывается время компоновки, а не компиляции.

Раздельная компиляция в языке Fortran

Когда был разработан Fortran, программы вводились в компьютер с помощью перфокарт, и не было никаких дисков или библиотек программ, которые известны сегодня.

Компилируемый модуль в языке Fortran идентичен выполняемому модулю, а именно подпрограмме, называемой *сабрутиной (subroutine).* Каждая сабрутина компилируется не только раздельно, но и *независимо,* и в результате одной компиляции не сохраняется никакой информации, которую можно использовать при последующих компиляциях.

Это означает, что не делается абсолютно никакой проверки на соответствие формальных и фактических параметров. Вы можете задать значение с плавающей точкой для целочисленного параметра. Более того, массив передается как указатель на первый элемент, и вызванная подпрограмма никак не может узнать размер массива или даже тип элементов. Подпрограмма может даже попытаться обратиться к несуществующему фактическому параметру. Другими словами, согласование формальных и фактических параметров — задача программиста; именно он должен обеспечить, правильные объявления типов и размеров параметров, как в вызывающих, так и вызываемых подпрограммах.

Поскольку каждая подпрограмма компилируется независимо, нельзя совместно использовать глобальные объявления данных. Вместо этого определены *общие (common) блоки:*

```
subroutine S1
common /block1/distance(100), speed(100), time(100)
real distance, speed, time
…
end
```

Это объявление требует выделить 300 ячеек памяти для значений с плавающей точкой. Все другие объявления для этого же блока распределяются в те же самые ячейки памяти, поэтому, если другая подпрограмма объявляет:

```
subroutine S2
common /block1/speed(200), time(200), distance(200)
integer speed, time, distance
….
End
```

то две подпрограммы будут использовать различные имена и различные типы для доступа к одной и той же памяти! Отображение common-блоков друг на друга делается по их *расположению* в памяти, а не по *именам переменных.* Если для переменной типа real выделяется столько памяти, сколько для двух переменных типа integer, speed(8O) в подпрограмме S2 размещается в той же самой памяти, что и половина переменной distance(40) в S1. Эффект подобен неаккуратному использованию типов union в языке C или вариантных записей в языке Pascal.

Независимая компиляция и общие блоки вряд ли создадут проблемы для отдельного программиста, который пишет небольшую программу, но с большой вероятностью вызовут проблемы в группе из десяти человек; придется организовывать встречи или контроль, чтобы гарантировать, что интерфейсы реализованы правильно. Частичное решение состоит в том, чтобы использовать включаемые (include) файлы, особенно для общих блоков, но вам все равно придется проверять, что вы используете последнюю версию включаемого файла, и удостовериться, что какой-нибудь умный программист не игнорирует объявления в файле.

Раздельная компиляция в языке C

Язык C отличается от других языков программирования тем, что понятие *файла* с исходным кодом появляется в определении языка и, что существенно, в терминах области действия и видимости идентификаторов. Язык C поощряет раздельную компиляцию до такой степени, что по умолчанию к каждой подпрограмме и каждой глобальной переменной можно обращаться отовсюду в программе.

Вначале немного терминологии: *объявление* вводит имя в программу:

```
void proc(void);
```

Имя может иметь много (идентичных) объявлений, но только одно из них будет также и *определением,* которое создает объект этого имени: отводит память для переменных или задает реализацию подпрограммы.

Следующий файл содержит главную программу main, а также *определение* глобальной переменной и *объявление* функции, имена которых по умолчанию подлежат внешнему связыванию:

```
        /* File main.c */
int global;                     /* Внешняя по умолчанию */
int func(int);                  /* Внешняя по умолчанию */

int main(void)
{
    global = 4;
     return func(global);
}
```

В отдельном файле дается определение (реализация) функции; переменная global объявляется снова, чтобы функция имела возможность к ней обратиться:

```
        /* File func.c */
extern int global;                          /* Внешняя, только объявление */

int func(int parm)
{
    return parm + global:
}
```

Обратите внимание, что еще одно объявление func не нужно, потому что определение функции в этом файле служит также и объявлением, и по умолчанию она внешняя. Однако для того чтобы func имела доступ к глобальной переменной, объявление переменной дать необходимо, и должен использоваться спецификатор extern. Если extern не используется, объявление переменной global будет восприниматься как второе определение переменной. Произойдет ошибка компоновки, так как в программе запрещено иметь два определения для одной и той же глобальной переменной.

Компиляция в языке С независима в том смысле, что результат одной компиляции не сохраняется для использования в другой. Если кто-то из вашей группы случайно напишет:

```
        /* File func.c */
extern float global;                    /* Внешняя, только объявление */
int func(int parm)                      /* Внешняя по умолчанию */
{
return parm + global;
}
```

программа все еще может быть откомпилирована и скомпонована, а ошибка произойдет только во время выполнения. На моем компьютере целочисленное значение 4, присвоенное переменной global в main, воспринимается в файле func.c как очень малое число с плавающей точкой; после обратного преобразования к целому числу оно становится нулем, и функция возвращает 4, а не 8.

Как и в языке Fortran, проблему можно частично решить, используя включаемые файлы так, чтобы одни и те же объявления использовались во всех файлах. И объявление extern для функции или переменной, и определение могут появиться в одном и том же вычислении. Поэтому мы помещаем все внешние объявления в один или несколько включаемых файлов, в то время как единственное определение для каждой функции или переменной будет содержаться не более чем в одном файле «.c»:

```
          /* File main.h */
extern int global;                      /* Только объявление */
          /* File func.h */
extern int func(int parm);              /* Только объявление */
          /* File main.c */
#include "main.h"
#include "func.h"
int global;                             /* Определение */
int main(void)
{
    return func(global) + 7;
}
        /* File func.c */
#include "main.h"
#include "func.h"
int func(int parm)                      /* Определение */
{
    return parm + global;
}
```

Спецификатор static

Забегая вперед, мы теперь покажем, как в языке С можно использовать свойства декомпозиции для имитации конструкции модуля других языков. В файле, содержащем десятки глобальных переменных и определений подпрограмм, обычно только некоторые из них должны быть доступны вне файла. Каждому определению, которое *не* используется внешним образом, должен предшествовать спецификатор static (статический), который указывает компилятору, что объявленная переменная или подпрограмма известна только внутри файла:

```
static      int g 1;                    /* Глобальная переменная только в этом файле */
            int g2;                     /* Глобальная переменная для всех файлов */
static      int f1 (int i) {...};       /* Глобальная функция только в этом файле */
            intf2(int i) {...};         /* Глобальная функция для всех файлов */
```

Здесь уместно говорить об *области действия файла (file scope)*, которая выступает в роли области действия модуля (module scope), используемой в других языках. Было бы, конечно, лучше, если бы по умолчанию принимался спецификатор static, а не extern; однако нетрудно привыкнуть приписывать к каждому глобальному объявлению static.

Источником недоразумений в языке С является тот факт, что static имеет другое значение, а именно он определяет, что время жизни переменной является всем временем выполнения программы. Как мы обсуждали в разделе 7.4, локальные переменные внутри процедуры имеют время жизни, ограниченное одним вызовом процедуры. Глобальные переменные, однако, имеют *статическое* время жизни, то есть они распределяются, когда программа начинается, и не освобождаются, пока программа не завершится. Статическое время жизни — нормальный режим для глобальных переменных; на самом деле, глобальные переменные, объявленные с extern, также имеют статическое время жизни!

Спецификатор static также можно использовать для локальных переменных, чтобы задать статическое время жизни:

```
void proc(void)
{
   static bool first_time = true;
    if (first_time) {
          /* Операторы, выполняемые при первом вызове proc */
          first_time = false;
    }
    ….
}
```

Подведем итог: все глобальные переменные и подпрограммы в файле должны быть объявлены как static, если явно не требуется, чтобы они были доступны вне файла. В противном случае они должны быть определены в одном файле без какого-либо спецификатора и экспортироваться через объявление их во включаемом файле со спецификатором extern.

13.2. Почему необходимы модули?

В предыдущем разделе мы рассматривали декомпозицию программ с чисто механической точки зрения, исходя из желания раздельно редактировать и компилировать части программы в разных файлах. Начиная с этого раздела мы обсудим декомпозицию программы на компоненты, возникающие в соответствии со смысловой структурой проекта и, может быть, кроме того допускающие раздельную компиляцию. Но сначала давайте спросим, почему декомпозиция так необходима?

Вам, возможно, объясняли, что человеческий мозг в любой момент времени способен иметь дело только с небольшим объемом материала. В терминах программирования это обычно выражается в виде требования, чтобы отдельная подпрограмма была не больше одной «страницы». Считается, что подпрограмма является концептуальной единицей: последовательностью операторов, выполняющих некоторую функцию. Если подпрограмма достаточно мала, скажем от 25 до 100 строк, можно легко понять все связи между составляющими ее операторами.

Но, чтобы понять всю программу, мы должны понять связи между подпрограммами, которые ее составляют. По аналогии должны быть понятны программы, содержащие от 25 до 100 подпрограмм, что составляет от 625 до 10000 строк. Такой размер программ относительно невелик по сравнению с промышленными и коммерческими программными системами, содержащими 100000, если не миллион, строк. Опыт показывает, что 10000 строк, возможно, является верхним пределом для размера монолитной программы и что необходим новый механизм структурирования, чтобы создавать и поддерживать большие программные системы.

Стандартным термином для механизма структурирования больших программ является *модуль (module),* хотя два языка, на которых мы сосредоточили внимание, используют другие термины: *пакеты (packages)* в языке Ada и *классы (classes) в* языке C++. В стандарте языка Pascal не определено никакого метода раздельной компиляции или декомпозиции программ. Например, первый Pascal-компилятор был *единой* программой, содержащей свыше 8000 строк кода на языке Pascal. Вместо того чтобы изменять Pascal, Вирт разработал новый (хотя и похожий) язык, названный Modula, так как центральным понятием в нем является модуль. К сожалению, многие поставщики расширили язык Pascal несовместимыми модульными конструкциями, поэтому Pascal не годится для написания переносимого программного обеспечения. Поскольку модули очень важны для разработки программного обеспечения, мы сосредоточим обсуждение на языке Ada, в котором разработана изящная модульная конструкция — так называемые *пакеты*.

13.3. Пакеты в языке Ada

Основной идеей, лежащей в основе модулей вообще и пакетов Ada в частности, является то, что такие вычислительные ресурсы, как данные и подпрограммы, должны быть *инкапсулированы* в некий единый модуль. Доступ к компонентам модуля разрешается только в соответствии с явно специфицированным интерфейсом. На рисунке 13.1 показана графическая запись (называемая *диаграммой Буча — Бухера),* применяемая в разработках на языке Ada.

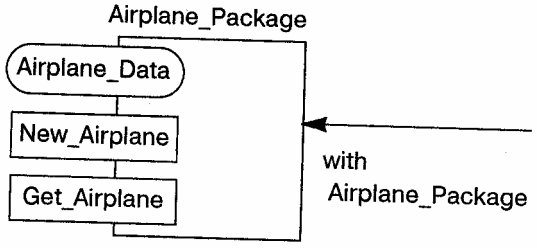

Рис. 13.1. Диаграмма пакета.

Большой прямоугольник обозначает пакет Airplane_Package, содержащий скрытые вычислительные ресурсы, а малые прямоугольники — окна, которые дают пользователю пакета доступ к скрытым ресурсам, овал обозначает, что экспортируется тип; а два прямоугольника — что экспортируются подпрограммы. Из каждого модуля, использующего ресурсы пакета, выходит стрелка, которая указывает на пакет.

Объявление пакета

Пакет состоит из двух частей: *спецификации* и *тела*. Тело инкапсулирует вычислительные ресурсы, а спецификация определяет интерфейс для этих ресурсов. Пакет из следующего примера предназначается для представления компонента системы управления воздушным движением, который хранит описание всех самолетов в контролируемом воздушном пространстве. Спецификация пакета объявляет тип и две подпрограммы интерфейса:

```
package Airplane_Package is
        type Airplane_Data is
            record
                ID:String(1 ..80);
                Speed: Integer range 0.. 1000;
                Altitude: Integer range 0..100;
            end record;
          procedure New_Airplane(Data: in Airplane_Data; I: out Integer);
          procedure Get_Airplane(l: in Integer; Data: out Airplane_Data);
end Airplane_Package;
```

Спецификация пакета содержит не тела, а только объявления процедур, заканчивающиеся точкой с запятой и вводимые зарезервированным словом is. Объявление служит только в качестве спецификации вычислительного ресурса, который предоставляет пакет.

В теле пакета должны быть обеспечены все ресурсы, которые были заявлены. В частности, для каждого объявления подпрограммы должно существовать тело подпрограммы с точно тем же самым объявлением:

```
package body Airplane_Package is
   Airplanes: array(1..1000) of Airplane_Data;
   Current_Airplanes: Integer range O..Airplanes'Last;

   function Find_Empty_Entry return Integer is
   begin
   …
   end Find_Empty_Entry;

     procedure New_Airplane(Data: in Airplane_Data; I: out Integer) is
           Index: Integer := Find_Empty_Entry;
     begin
        Airplanes(lndex) := Data;
         I := Index;
      end New_Airplane;

       procedure Get_Airplane(l: in Integer; Data: out Airplane_Data) is
       begin
            Data := Airplanes(l);
       end Get_Airplane;
end Airplane_Package;
```

Чего мы добились? Структура, применяемая для хранения данных о самолетах (здесь это массив фиксированного размера), инкапсулирована в тело пакета. Правило языка Ada состоит в том, что изменение в теле пакета не требует изменений ни спецификации пакета, ни любого другого компонента программы, использующего пакет. Более того, не нужно даже их перекомпилировать. Например, если впоследствии вы должны заменить массив связанным списком, не нужно изменять никаких других компонентов системы *при условии*, что интерфейс, описанный в спецификации пакета, не изменился:

```
package body Airplane_Package is
    type Node;
    type Ptr is access Node;
    type Node is
       record
       Info: Airplane_Data;
       Next: Ptr;
       end record;

  Head: Ptr; .                          -- Начало связанного списка

procedure New_Airplane(Data: in Airplane_Data; I: out Integer) is
begin
   …                                    -- Новая реализация
end New_Airplane;

procedure Get_Airplane(l: in Integer; Data: out Airplane_Data) is
begin
  …                                     -- Новая реализация
end Get_Airplane;
end Airplane_Package;
```

Инкапсуляция делается не только для удобства, но и для надежности. Пользователям пакета не разрешен непосредственный доступ к данным или внутренним подпрограммам (таким, как Find_Empty_Entry) тела пакета. Таким образом, никакой другой программист из группы не может случайно (или преднамеренно) изменить структуру данных способом, который не был предусмотрен. Ошибка в реализации пакета обязательно локализована внутри кода тела пакета и не является результатом некоторого кода, написанного членом группы, не ответственным за пакет.

Спецификация и тело пакета — это разные модули, и их можно компилировать раздельно. Однако в терминах объявлений они рассматриваются как одна область действия, например, тип Airplain_Data известен внутри тела пакета. Это означает, конечно, что спецификация должна компилироваться перед телом. В отличие от языка С, здесь нет никакого понятия «файла», и объявления в языке Ada существуют только внутри такой единицы, как подпрограмма или пакет. Несколько компилируемых модулей могут находиться в одном файле, хотя обычно удобнее хранить каждый модуль в отдельном файле.

Соглашение для написания программ на языке С, предложенное в предыдущем разделе, пытается имитировать инкапсуляцию, которая предоставляется пакетами в языке Ada. Включаемые файлы, содержащие внешние объявления, соответствуют спецификациям пакета и с помощью записи static для всех глобальных переменных и подпрограмм в файле достигается эффект тела пакета. Конечно, это всего лишь «бюрократический» прием, и его легко обой-ти, но это хороший способ структурирования программ в языке С.

Использование пакета

Программа на языке Ada (или другой пакет) может получить доступ к вычис- лительным ресурсам пакета, задав *контекст (context clause)* перед первой строкой программы:

```
with Airplane_Package;
procedure Air_Traffic_Control is
      A: Airplane_Package.Airplane_Data;
       Index: Integer;
begin
   while... loop
        A :=...;                              -- Создать запись
        Airplane_Package. New_Airplane(A, Index):
                                              -- Сохранить в структуре данных
     end loop;
end Air_Traffic_Control;
```

With-конструкция сообщает компилятору, что эта программа должна компилироваться в среде, которая включает все объявления пакета Airplain_Package. Синтаксис для именования компонентов пакета аналогичен синтаксису для выбора компонентов записи. Поскольку каждый пакет должен иметь уникальное имя, компоненты в разных пакетах могут иметь одинаковые имена, и никакого конфликта не возникнет. Это означает, что управление *пространством имен,* т. е. набором имен, в программном проекте упрощено, и необходимо осуществлять контроль только на уровне имен пакетов. Сравните это с языком С, где идентификатор, который экспортируется из файла, видим во всех других файлах, потому недостаточно только обеспечить различие имен файлов.

With-конструкция добавляет составные имена к пространству имен компиляции; также можно включить use-конструкцию, чтобы открыть пространство имен и разрешить прямое именование компонентов, встречающихся в спецификации:

```
with Airplane_Package;
use Airplane_Package;
procedure Air_Traffic_Control is
      A: Airplane_Data;                       -- Непосредственно видима
      Index: Integer; begin
      New_Airplane(A, Index):                 -- Непосредственно видима
end Air-Traffic-Control;
```

Одна трудность, связанная с use-конструкциями, состоит в том, что вы можете столкнуться с неоднозначностью, если use-конструкции для двух пакетов открывают одно и то же имя или если существует локальное объявление с тем же самым именем, что и в пакете. Правила языка определяют, каким в случае неоднозначности должен быть ответ компилятора.

Важнее, однако, то, что модуль, в котором with- и use-конструкции связаны с множеством пакетов, может стать практически нечитаемым. Такое имя, как Put_Element, могло бы исходить почти из любого пакета, в то время как местоположение Airplane_Package.Put_Element вполне очевидно. Ситуация аналогична программе, написанной на языке С, в которой много включаемых файлов: у вас просто нет удобного способа отыскивать объявления, и единственное решение — использовать внешний программный инструмент или соглашения о наименованиях.

Программистам, пишущим на языке Ada, следует использовать преимущества самодокументирования модулей за счет with, а use-конструкции применять только в небольших сегментах программы, где все вполне очевидно, а полная запись была бы чересчур утомительна. К счастью, можно поместить use-конструкции внутри локальной процедуры:

```
procedure Check_for_Collision is
    use Airplane_Package;
    A1: Airplane-Data;
begin
     Get_Airplane(1, A1);
end Check_for_Collision;
```

В большинстве языков программирования *импортирующий* модуль автоматически получает все общие (public) ресурсы импортированного модуля. В некоторых языках, подобных языку Modula, импортирующему модулю разрешается точно определять, какие ресурсы ему требуются. Этот метод позволяет избежать перегрузки пространства имен, вызванной включающим характером use-конструкции в языке Ada.

Порядок компиляции

with-конструкции определяют естественный порядок компиляции: спецификация пакета должна компилироваться *перед телом* и *перед* любым модулем, который связан с ней через with. Однако упорядочение является частичным, т. е. порядок компиляции тела пакета и единиц, которые используют пакет, может быть любым. Вы можете исправить ошибку в теле пакета или в использующей его единице, перекомпилировав только то, что изменилось, но изменение спецификации пакета требует перекомпиляции как тела, так и всех использующих его единиц. В очень большом проекте следует избегать изменений спецификации пакетов, потому что они могут вызвать лавину перекомпиляций: P1 используется в P2, который используется в P3, и т. д.

Тот факт, что компиляция одной единицы требует результатов компиляции других единиц, означает, что в языке Ada компилятор должен содержать *библиотеку* для хранения результатов компиляции. Библиотека может быть просто каталогом, содержащим порожденные файлы, или сложной базой данных. При использовании любого метода библиотечный администратор является центральным компонентом реализации языка Ada, а не просто необязательным программным инструментом. Библиотечный администратор языка Ada проводит в жизнь правило, согласно которому при изменении спецификации пакета необходимо перекомпилировать тело и использующие его единицы. Таким образом, компилятор языка Ada уже включает инструмент сборки программы (make) с перекомпиляцией измененных модулей, который в других средах программирования является необязательной утилитой, а не частью языковых средств.

13.4. Абстрактные типы данных в языке Ada

Airplane_Package — это *абстрактный объект данных*. Он является *абстрактным*, потому что пользователь пакета не знает, реализована ли база данных самолетов как массив, список или как дерево. Доступ к базе данных осуществляется только через объявленные в спецификации пакета интерфейсные процедуры, которые позволяют пользователю абстрактно создавать и отыскивать значение типа Airplane_Data, не зная, в каком виде оно хранится.

Пакет является *объектом данных*, потому что он действительно содержит данные: массив и любые другие переменные, объявленные в теле пакета. Правильно рассматривать Airplane_Package как особую* переменную: для нее должна быть выделена память и есть некоторые операции, которые могут изменить ее значение. Это объект не первого класса", потому что он не имеет всех преимуществ обычных переменных: нельзя делать присваивание пакету или передавать пакет как параметр.

Предположим теперь, что мы нуждаемся в *двух* таких базах данных: одна для смоделированного пульта управления воздушным движением и одна для администратора сценария моделирования, который вводит и инициализирует новые самолеты. Можно было бы написать два пакета с незначительно отличающимися именами или написать родовой пакет и дважды его конкретизировать, но это очень ограниченные решения. Что мы действительно хотели бы сделать, так это объявить столько таких объектов, сколько нам нужно, так же как мы объявляем целые числа. Другими словами, мы хотим иметь возможность конструировать *абстрактный тип данных* (Abstract Data Type — ADT), который является точно таким же, как и абстрактный объект данных, за исключением того что он не содержит никаких «переменных». Вместо этого, подобно другим типам, ADT определяет набор значений и набор операций на этих значениях, а фактическое объявление переменных этого типа может быть сделано в других компонентах программы.

ADT в языке Ada — это пакет, который содержит только объявления констант, типов и подпрограмм. Спецификация пакета включает объявление типа так, что другие единицы могут объявлять один или несколько объектов типа Airplains (самолеты):

```
package Airplane_Package is
   type Airplane_Data is ... end record;
   type Airplanes is
      record
           Database: array( 1.. 1000) of Airplane_Data;
           Current_Airplanes: Integer O..Database'Last;
      end record;
procedure New_Airplane(
     A: in out Airplanes; Data: in Airplane_Data: I: out Integer);
procedure Get_Airplane(
          A: in out Airplanes; I: in Integer; Data: out Airplane_Data);
end Airplane_Package;
```

Тело пакета такое же, как и раньше, за исключением того что в нем нет никаких глобальных переменных:

```
package body Airplane_Package is
      function Find_Empty_Entry... ;
       procedure New_Airplane...;
       procedure Get_Airplane...;
end Airplane_Package;
```

Программа, которая использует пакет, может теперь объявить одну или несколько переменных типа, поставляемого пакетом. Фактически тип является обычным типом и может использоваться в последующих определениях типов и как тип параметра:

```
with Airplane_Package;
procedure Air_Traffic_Control is
  Airplane: Airplane_Package.Airplanes;
                        -- Переменная ADT
   type Ptr is access Airplane_Package.Airplanes;
                        -- Тип с компонентом ADT
    procedure Display(Parm: in Airplane_Package.Airplanes);
                        -- Параметр ADT
    A: Airplane_Package.Airplane_Data;
    Index: Integer;
begin
A .:=... ;
Airplane_Package.New_Airplane(Airplane, A, Index);
Display(Airplane);
```

```
end Air_Traffic_Control;
```

За использование ADT вместо абстрактных объектов данных придется заплатить определенную цену: так как в теле пакета больше нет *ни одного* неявного объекта, каждая интерфейсная процедура должна содержать дополнительный параметр, который явно сообщает подпрограмме, какой именно объект нужно обработать.

Вы можете спросить: а как насчет «абстракции»? Поскольку тип Airplaines теперь объявлен в спецификации пакета, мы потеряли все абстракции; больше нельзя изменить структуру данных, не повлияв на другие единицы, использующие пакет. Кроме того, кто-нибудь из группы программистов может скрытно проигнорировать процедуры интерфейса и написать «улучшенный» интерфейс. Мы должны найти решение, в котором имя типа находится в спецификации так, чтобы его можно было использовать, а детали реализации инкапсулированы — что-нибудь вроде следующего:

```
package Airplane_Package is
   type Airplane_Data is ... end record;
   type Airplanes;                                  -- Неполное объявление типа
end Airplane_Package;

package body Airplane_Package is
        type Airplanes is                           -- Полное объявление типа
          record
            Database: array(1..1000) of Airplane_Data;
            Current_Airplanes: Integer 0...Database'Last;
          end record;
   …
end Airplane_Package;
```

Потратьте несколько минут, чтобы проанализировать этот вариант самостоятельно перед тем, как идти дальше.

Что касается пакета, то с этими объявлениями нет никаких проблем, потому что спецификация и тело формируют одну область объявлений. Проблемы начинаются, когда мы пробуем использовать пакет:

```
with Airplane_Package;
procedure Air_Traffic_Control is
      Airplane_1: Airplane_Package.Airplanes;
      Airplane_2: Airplane_Package.Airplanes;
…
end Air_Traffic_Control;
```

Язык Ada задуман так, что компиляции спецификации пакета достаточно, чтобы сделать возможной компиляцию любой единицы, использующей пакет. Фактически, не нужно даже, чтобы существовало тело пакета, когда компилируется использующая единица. Но чтобы откомпилировать приведенную выше программу, компилятор должен знать, сколько памяти нужно выделить для Airplane_1 и Airplane_2; аналогично, если эта переменная используется в выражении или передается как параметр, компилятор должен знать размер переменной. Таким образом, если представление ADT инкапсулировано в тело пакета, откомпилировать программу будет невозможно.

Приватные (private) типы

Поскольку мы имеем дело с реальными языками программирования, которые должны компилироваться, не остается ничего другого, кроме как вернуть полную спецификацию типа в спецификацию пакета. Чтобы достичь абстракции, используется комбинация самообмана и правил языка:

```
package Airplane_Package is
       type Airplane_Data is ... end record;
       type Airplanes is private;
                    -- Детали будут заданы позже
        procedure New_Airplane(Data: in Airplane_Data; I: out Integer);
        procedure Get_Airplane(I: in Integer; Data: out Airplane_Data);
private
   type Airplanes is                          -- Полное объявление типа
record
          Database: array(1 ..1000) of Airplane_Data;
          Current_Airplanes: Integer 0.. Database'Last;
end record;
end Airplane_Package;
```

Сам тип первоначально объявлен *как приватный (private)*, в то время как полное объявление типа записано в специальном разделе спецификации пакета, который вводится ключевым словом private. Тип данных абстрактный, потому что компилятор предписывает правило, по которому единицам, обращающимся к пакету через with, не разрешается иметь доступ к информации, записанной в закрытой (private) части. Им разрешается обращаться к приватному типу данных только через подпрограммы интерфейса в открытой (public) части спецификации; эти подпрограммы реализованы в теле, которое может иметь доступ к закрытой части. Так как исходный код использующих единиц не зависит от закрытой части, можно изменить объявления в закрытой части, не нарушая правильности исходных текстов использующих единиц; но, конечно, нужно будет сделать перекомпиляцию, потому что изменение в закрытой части могло привести к изменению выделяемого объема памяти. Поскольку вы не можете явно использовать информацию из закрытой части, вы должны «сделать вид», что не можете ее даже видеть. Например, нет смысла прикладывать особые усилия в написании чрезвычайно эффективных алгоритмов, зная, что приватный тип реализован как массив, а не как список, потому что руководитель проекта может, в конечном счете, изменить реализацию.

Ограниченные типы

Достаточно объявить объект (переменную или константу) приватного типа, и над ним можно будет выполнять операции присваивания и проверки на равенство, так как эти операции выполняются поразрядно независимо от внутренней структуры. Существует, однако, концептуальная проблема, связанная с разрешением присваивания и проверки равенства. Предположим, что в реализации массив заменен на указатель:

```
package Airplane_Package is
   type Airplanes is private;
…
private
   type Airplanes_jnfo is
    record
        Database: array(1..1000) of Airplane_Data;
       Current_Airplanes: Integer O..Database'Last;
```

```
    end record;
  type Airplanes is access Airplanes_info;
end Airplane_Package;
```

Мы обещали, что при изменении закрытой части не потребуется менять использующие единицы, но здесь это не так, потому что присваивание делается для указателей, а не для указуемых объектов:

```
with Airplane_Package;
procedure Air_Traffic_ControI is
Airplane_1: Airplane_Package.Airplanes;
Airplane_2: Airplane_Package.Airplanes;
begin
Airplane_1 := Airplane_2;              -- Присваивание указателей
end Air_Traffic_Control;
```

Если присваивание и проверка равенства не имеют смысла (например, при сравнении двух массивов, которые реализуют базы данных), язык Ada позволяет вам объявить приватный тип как ограниченный (limited). Объекты ограниченных типов нельзя присваивать или сравнивать, но вы можете явно написать свои собственные версии для этих операций. Это решит только что описанную проблему; при преобразовании между двумя реализациями можно изменить в теле пакета явный код для присваивания и равенства, чтобы гарантировать, что эти операции по-прежнему имеют смысл. Неограниченными приватными типами следует оставить лишь «небольшие» объекты, которые, вероятно, не подвергнутся другим изменениям, кроме добавления или изменения поля в записи.

Обратите внимание, что если приватный тип реализован с помощью указателя, то в предположении, что все указатели представлены одинаково, уже не важно, каков тип указуемого объекта. В языке Ada такое предположение фактически делается и, таким образом, указуемый тип может быть определен в теле пакета. Теперь изменение структуры данных благодаря косвенности доступа не требует даже перекомпиляции единиц с конструкцией with:

```
package Airplane_Package is
    type Airplanes is private;
…
private
type Airplanes_info;                    -- Незавершенное объявление типа
type Airplanes is access Airplanes_info;
end Airplane_Package;

package body Airplane_Package is
  type Airplanes_info is                -- Завершение в теле
record
      Database: array(1..1000) of Airplane_Data;
      Current_Airplanes: Integer O..Database'Last;
end record;
end Airplane_Package;
```

ADT является мощным средством структурирования программ благодаря четкому отделению спецификации от реализации:

• Используя ADT, можно делать серьезные изменения в отдельных компонентах программы надежно, не вызывая ошибок в других частях программы.

• ADT может использоваться как инструмент управления разработкой: архитектор проекта разрабатывает интерфейсы, а каждый член группы программистов реализует один или несколько ADT.

• Можно выполнить тестирование и частичную интеграцию, применяя вырожденные реализации отсутствующих тел пакетов.

В главе 14 мы подробнее поговорим о роли ADT как основы объектно-ориентированного программирования.

13.5. Как писать модули на языке C++

Язык C++ — это расширение языка C, и поэтому здесь тоже существует понятие файла как единицы структурирования программ. Наиболее важным расширением является введение *классов (classes)*, которые непосредственно реализуют абстрактные типы данных, в отличие от языка Ada, который использует комбинацию из двух понятий: пакета и приватного типа данных. В следующей главе мы обсудим объектно-ориентированное программирование, которое основано на классах; а в этом разделе объясним основные понятия классов и покажем, как они могут использоваться для определения модулей.

Класс аналогичен спецификации пакета, которая объявляет один или несколько приватных типов:

```
class Airplanes {
public:
   struct Airplane_Data {
          char id[80];
           int speed;
           int altitude;
      };
     void new_airplane(const Airplane_Data & a, int & i);
      void get_airplane(int i, Airplane_Data & a) const;
private:
  Airplane_Data database[1000];
   int current_airplanes;
   int find_empty_entry();
 };
```

Обратите внимание, что имя класса, которое является именем типа, также служит в качестве имени инкапсулирующей единицы; никакого самостоятельного имени модуля не существует. Класс имеет общую и закрытую части. По умолчанию компоненты класса являются приватными, поэтому перед общей частью необходим спецификатор public. Фактически, при помощи спецификаторов public и private можно задать несколько открытых и закрытых частей вперемежку, в отличие от языка Ada, который требует, чтобы для каждой части был только один список объявлений:

```
class C {
public:
…
private:
…
public:
….
private:
…..
 };
```

Объявления в общей части доступны любым модулям, использующим этот класс, в то время как объявления в закрытой части доступны только внутри класса. Спецификатор const *в* get_airplane — это следующее средство управления, он означает, что подпрограмма не изменяет никакие данные внутри объекта класса. Такие подпрограммы называются *инспекторами (inspectors).*

Поскольку класс является типом, могут быть объявлены объекты (константы и переменные) этого класса, так называемые *экземпляры класса:*

```
Airplanes Airplane;                    // Экземпляр класса Airplanes

int index;
Airplanes::Airplane_Data a;
Airplane.new_airplane(a, index);       // Вызов подпрограммы для экземпляра
```

Классом может быть и тип параметра. Для каждого экземпляра будет выделена память всем переменным, объявленным в классе, точно так же, как для переменной типа запись выделяется память всем полям.

Синтаксис вызова подпрограммы отличается от синтаксиса, принятого в языке Ada, из-за различий в исходных концепциях. Вызов в языке Ada:

```
Airplane_Package.New_Airplane(Airplane, A, Index);
```

рассматривает пакет как применение ресурса — процедуры New_Airplane, которой должен быть задан конкретный объект Airplane. Язык C++ полагает, что объект Airplane — это экземпляр класса Airplanes, и, если вы посылаете объекту *сообщение (message)* new_airplane, для этого объекта будет выполнена соответствующая процедура.

Обратите внимание, что даже такие подпрограммы, как find_empty_entry, которые используются только внутри класса, объявлены в определении класса. Язык C++ не имеет ничего похожего на тело пакета, представляющее собой единицу, которая инкапсулирует реализацию интерфейса и других подпрограмм. Конечно, внутренняя подпрограмма недоступна другим модулям, потому что она объявлена внутри закрытой части. В языке C++ проблема состоит в том, что, если необходимо изменить объявление find_empty_entry или добавить другую приватную подпрограмму, придется перекомпилировать все модули программы, которые используют этот класс; в языке Ada изменение тела пакета не воздействует на остальную часть программы. Чтобы достичь на языке C++ реального разделения интерфейса и реализации, следует объявить интерфейс как абстрактный класс, а затем получить конкретный производный класс, который содержит реализацию (см. раздел 15.1).

Где находятся подпрограммы реализованного класса? Ответ состоит в том, что они могут быть реализованы где угодно, в частности в отдельном файле, который обращается к определению класса через включаемый файл. Операция разрешения контекста «::» идентифицирует каждую подпрограмму как принадлежащую конкретному классу:

```
            // Некоторый файл
#include "Airplanes.h"                 // Содержит объявление класса

void Airplanes::new_airplane(const Airplane_Data & a, int & i)
{
…
}
void Airplanes::get_airplane(int i, Airplane_Data & a) const
{
….
}
int Airplanes::find_empty_entry()
{
```

…
}

Обратите внимание, что внутренняя подпрограмма find_empty_entry должна быть объявлена внутри (в закрытой части) класса так, чтобы она могла обращаться к приватным данным.

Пространство имен

Одним из последних добавлений к определению языка C++ была конструкция namespace *(пространство имен),* которая дает возможность программисту ограничить область действия других глобальных объектов так же, как это делается с помощью пакета в языке Ada. Конструкция, аналогичная use-предложению в Ada, открывает пространство имен:

```
namespace N1 {
    void proc();                          //Процедура в пространстве имен
};
namespace N2 {
void proc();                              // Другая процедура
};
N1:: proc(),                              //Операция разрешения контекста для доступа
using namespace N1 ;
proc();                                   // Правильно
using namespace N2;
proc();                                   //Теперь неоднозначно
```

К сожалению, в языке C++ не определен библиотечный механизм: объявления класса могут использоваться совместно только через включаемые файлы. Группа разработчиков должна организовать процедуры для обновления включаемых файлов, отдавая предпочтение программным инструментальным средствам, чтобы оповещать членов группы о том, что две компиляции не используют одну и ту же версию включаемого файла.

13.6. Упражнения

1. Напишите главную программу на языке C, которая вызывает внешнюю функцию f с целочисленным параметром; в другом файле напишите функцию f с параметром с плавающей точкой, который она печатает. Откомпилируйте, скомпонуйте и выполните программу. Что она печатает? Попытайтесь откомпилировать, скомпоновать и выполнить *ту же* самую программу на языке C++ .

2. Напишите программу, реализующую абстрактный тип данных для очереди, и главную программу, которая объявляет и использует несколько очередей. Очередь должна быть реализована как массив, который объявлен в закрытой части пакета языка Ada или класса C++. Затем измените реализацию на связанный список; главная программа должна выполняться без изменений.

3. Что происходит, если вы пытаетесь присвоить одну очередь другой? Решите проблему, используя ограниченный приватный тип в языке Ada или *конструктор копий (copy-constructor)* в C++.

4. В языках C и C++ в объявлении подпрограммы имена параметров не обязательны:

C

```
int func(int, float, char*);
```

Почему это так? Будут ли так или иначе использоваться имена параметров? Почему в языке Ada требуется, чтобы в спецификации пакета присутствовали имена параметров?

5. В языке Ada есть конструкция для раздельной компиляции, которая не зависит от конструкции пакета:

Ada

```
procedure Main is
     Global: Integer;
      procedure R is separate;                -- Раздельно компилируемая процедура
end Main;

separate(Main)                                --Другой файл
procedure R is
begin
Global := 4;                                  -- Обычные правила области действия
end R:
```

Факт раздельной компиляции локального пакета или тела процедуры не влияет на область действия и видимость. Как это может быть реализовано? Требуют ли изменения в раздельно компилируемой единице перекомпиляции родительской единицы? Почему? Обратный вопрос: как изменения в родителе воздействуют на раздельно компилируемую единицу?

6. Раздельно компилируемая единица может содержать конструкцию, задающую контекст:

Ada

```
with Text_IO;
separate(Main)
procedure R is
…
end R;
```

Как это можно использовать?

7. Следующая программа на языке Ada не компилируется; почему?

Ada

```
package P is
     type T is (A, B, C, D);
end P;

with P;
procedure Main is
    X: P.T;
begin
   if X = P. A then ...end if;
end Main;
```

Существуют четыре способа решить проблему; каковы преимущества и недостатки каждого из них: а) use-конструкция, б) префиксная запись, в) renames (переименование), г) конструкция use type в языке Ada 95?

Глава 14

Объектно-ориентированное программирование

14.1. Объектно-ориентированное проектирование

В предыдущей главе обсуждалась языковая поддержка структурирования программ, но мы не пытались ответить на вопрос: как следует разбивать программы на модули? Обычно этот предмет изучается в курсе по разработке программного обеспечения, но один метод декомпозиции программ, называемый *объектно-ориентированным программированием* (ООП), настолько важен, что современные языки программирования непосредственно поддерживают этот метод. Следующие две главы будут посвящены теме языковой поддержки ООП.

При проектировании программы естественный подход должен состоять в том, чтобы исследовать требования в терминах функций или операций, то есть задать вопрос: что должна делать программа? Например, программное обеспечение для предварительной продажи билетов в авиакомпании должно выполнять такие функции:

1. Принять от кассира место назначения заказчика и дату отправления.

2. Отобразить на терминале кассира список доступных рейсов.

3. Принять от кассира предварительный заказ на конкретный рейс.

4. Подтвердить предварительный заказ и напечатать билет.

Эти требования, естественно, находят отражение в проекте, показанном на рис. 14.1, с модулем для каждой функции и «главным» модулем, который вызывает другие.

К сожалению, этот проект не будет *надежным* в эксплуатации; даже для небольших изменений в требованиях могут понадобиться значительные изменения программного обеспечения. Для примера предположим, что авиакомпания улучшает условия труда, заменяя устаревшие дисплейные терминалы. Вполне правдоподобно, что для новых терминалов потребуется изменить все четыре модуля; точно так же придется вносить много исправлений, если изменятся соглашения о форматах информации, используемой совместно с другими компаниями.

Но все мы знаем, что изменение программного обеспечения чревато внесением ошибок; не устойчивый к ошибкам проект приведет к тому, что поставленная программная система будет ненадежной и неустойчивой. Вы могли бы возразить, что персонал должен воздержаться от изменения программного обеспечения, но весь смысл программного обеспечения состоит в том, что это именно *программное* обеспечение, а значит, его можно перепрограммировать, изменить; иначе все прикладные программы было бы эффективнее «зашить» подобно программе карманного калькулятора.

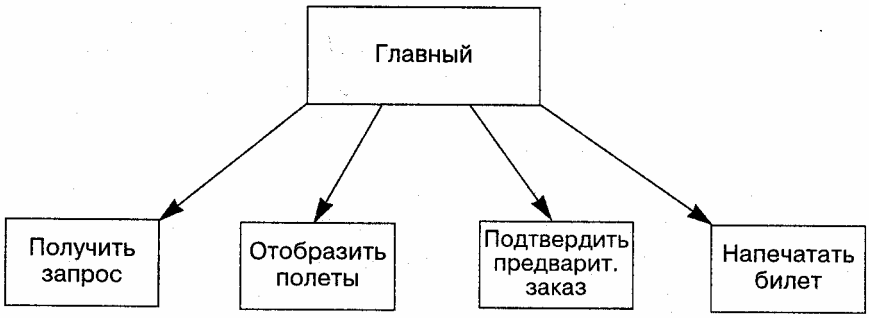

Рис. 14.1. Функциональная декомпозиция.

Программное обеспечение можно сделать намного устойчивее к ошибкам и надежнее, если изменить основные критерии, которыми мы руководствуемся при проектировании. Правильнее задать вопрос: *над чем* работает программное обеспечение? Акцент делается не на функциональных возможностях, а на внешних устройствах, внутренних структурах данных и моделях реального мира, т. е. на том, что принято называть *объектами (objects).* Модуль должен быть создан для каждого «объекта» и содержать все данные и операции, необходимые для реализации объекта. В нашем примере мы можем выделить несколько объектов, как показано на рис. 14.2.
Такие внешние устройства, как дисплейный терминал и принтер, идентифицированы как объекты, так же как и базы данных с информацией о рейсах и предварительных заказах. Кроме того, мы выделили объект Заказчик, назначение которого — моделировать воображаемую форму, в которую кассир вводит данные до того, как подтвержден рейс и выдан билет. Этот проект устойчив к ошибкам при внесении изменений:

Рис. 14.2. Объектно-ориентированное проектирование.

• Изменения, которые вносят для того, чтобы использовать разные терминалы, могут быть ограничены объектом Терминал. Программы этого объекта отображают данные заказчика на реальный дисплей и команды клавиатуры, так что объект Заказчик не должен изменяться, а только отображаться на новые аппаратные средства.

• Перераспределение кодов авиакомпаний может, конечно, потребовать общей реорганизации базы данных, но что касается остальных частей программы, то для них один двухсимвольный код авиакомпании ничем не отличается от другого.

Объектно-ориентированное проектирование можно использовать не только для моделирования реальных объектов, но и для создания многократно используемых программных компонентов. Это непосредственно связано с одной из концепций языков программирования, которую мы подчеркивали, — абстрагированием. Модули, реализующие структуры данных, могут быть разработаны и запрограммированы как объекты, которые являются экземплярами абстрактного типа данных вместе с операциями для обработки данных. Абстрагирование достигается за счет того, что представление типа данных скрывается внутри объекта.

Фактически, основное различие между объектно-ориентированным и «обычным» программированием состоит в том, что в обычном программировании мы ограничены встроенными абстракциями, в то время как в объектно-ориентированном мы можем

определять свои собственные абстракции. Например, числа с плавающей точкой (см. гл. 9) — это ничто иное, как удобная абстракция сложной обработки данных на компьютере. Хорошо было бы, если бы все языки программирования содержали встроенные абстракции для каждого объекта, который нам когда-нибудь понадобится (комплексные числа, рациональные числа, векторы, матрицы и т. д. и т. п.), но полезным абстракциям нет предела. В конечном счете, язык программирования нужно чем-то ограничить и оставить работу для программиста.

Как программист может создавать новые абстракции? Один из способов состоит в том, чтобы использовать соглашения кодирования и документирование («первый элемент массива — вещественная часть, а второй — мнимая часть»). С другой стороны, язык может обеспечивать такую конструкцию, как приватные типы в языке Ada, которая дает возможность программисту явно определить новые абстракции; эти абстракции будут компилироваться и проверяться точно так же, как и встроенные абстракции. ООП можно (и полезно) применять и в рамках обычных языков, но, аналогично другим идеям в программировании, оно работает лучше всего, когда используются языки, которые непосредственно поддерживают это понятие. Основная конструкция для поддержки ООП — абстрактный тип данных, который обсуждался в предыдущей главе, но важно понять, что объектно-ориентированное проектирование является более общим и простирается до абстрагирования внешних устройств, моделей реального мира и т. д.

Объектно-ориентированное проектирование — дело чрезвычайно слож-ное. Нужны большой опыт и здравый смысл, чтобы решить, что же заслуживает того, чтобы стать объектом. Новички в объектно-ориентированном проектировании склонны впадать в излишний энтузиазм и делать объектами буквально все; а это приводит к таким перегруженным и длинным утомительным программам, что теряются все преимущества метода. Наилучшее интуитивное правило, на которое стоит опираться, — это правило *упрятывания информации:*

В каждом объекте должно скрываться одно важное проектное решение.

Очень полезно бывает задать себе вопрос: «возможно ли, что это решение изменится за время жизни программы?»

Конкретные дисплейные терминалы и принтеры, выбранные для системы предварительных заказов, явно подлежат обновлению. Точно так же решения по организации базы данных, вероятно, будут изменяться, чтобы улучшить эффективность, поскольку система растет. С другой стороны, можно было бы привести доводы, что изменение формы данных заказчика маловероятно и что отдельный объект здесь не нужен. Даже если вы не согласны с нашим проектным решением создать объект Заказчик, вы должны согласиться, что объектно-ориентированное проектирование — хороший общий подход для обсуждения проблем разработки и достоинств одного проекта перед другим.

В следующих разделах языковая поддержка ООП будет обсуждаться на примере двух языков: C++ и Ada 95. Сначала мы рассмотрим язык C++, который был разработан как добавление одной интегрированной конструкции для ООП к языку C, в котором нет поддержки даже для модулей. Затем мы увидим, как полное объектно-ориентированное программирование определено в языке Ada 95 путем добавления нескольких небольших конструкций к языку Ada 83, который уже имел много свойств, частично поддерживающих ООП.

14.2. Объектно-ориентированное программирование на языке C++

Говорят, что язык программирования поддерживает ООП, если он включает конструкции для:

• инкапсуляции и абстракции данных,

• наследования,

• динамического полиморфизма.

Позвольте нам вернуться к обсуждению инкапсуляции и абстракции данных из предыдущей главы.

Такие модули, как пакеты в языке Ada, инкапсулируют вычислительные ресурсы, выставляя только спецификацию интерфейса. Абстракция данных может быть достигнута через определение представления данных в закрытой части, к которой нельзя обращаться из других единиц. Единица инкапсуляции и абстракции в языке C++ — это *класс (class),* который содержит объявления подпрограмм и типов данных. Из класса создаются фактические объекты, называемые экземлярами*(instances).* Пример класса в языке C++:

```
class Airplane_Data {
public:
    char *get_id(char *s) const          {return id;}
    void set_id(char *s)                 {strcpy(id, s);}
     int get_speed() const               {return speed;}
       void set_speed(int i)             {speed=i;}
      int get_altitude() const           {return altitude;}
      void set_altitude(int i)           {altitude = i;}
 private:
        char id[80];
         int speed;
         int altitude;
};
```

Этот пример расширяет пример из предыдущей главы, создавая отдельный класс для данных о каждом самолете. Этот класс может теперь *использоваться* другим классом, например тем, который определяет структуру для хранения данных о многих самолетах:

```
class Airplanes {
 public:
    void New_Airplane(Airplane_Data, int &);
    void Get_Airplane(int, Airplane_Data &) const;
private:
   Airplane_Data database[100];
      int current_airplanes;
      int find_empty_entry();
};
```

Каждый класс разрабатывается для того, чтобы инкапсулировать набор объявлений данных. Объявления данных в закрытой части могут быть изменены без изменения программ, использующих этот класс и называющихся *клиентами (clients) класса,* хотя их и придется перекомпилировать. Класс имеет набор интерфейсных функций, которые извлекают и обновляют значения данных, внутренних по отношению к классу.

Вы можете задать вопрос, почему Airplane_Data лучше сделать отдельным классом, а не просто объявить обычной общей (public) записью. Это спорное проектное решение: данные должны быть скрыты в классе, если вы полагаете, что внутреннее представление может измениться. Например, вы можете знать, что один заказчик предпочитает измерять высоту в английских футах, тогда как другой предпочитает метры. Определяя отдельный класс для Airplane_Data, вы можете использовать то же самое программное обеспечение для обоих заказчиков и изменить только реализацию функций доступа.

За эту гибкость приходится платить определенную цену; *каждый* доступ к значению данных требует вызова подпрограммы:

```
Aircraft_Data a;                    // Экземпляр класса
int alt;
alt = a.get_altitud(e);             // Получить значение, скрытое в экземпляре
alt = (alt* 2)+ 1000;
a.set_altitude(alt);                // Вернуть значение в экземпляр
```

вместо простого оператора присваивания в случае, когда a общая (public) запись:

```
a.alt = (a.alt*2) + 1000;
```

Программирование может стать очень утомительным, а получающийся в результате код трудно читаемым, потому что функции доступа затеняют содержательные операции обработки. Таким образом, классы должны вводиться только тогда, когда можно получить явное преимущество от скрытия деталей реализации абстрактного типа данных.

Однако инкапсуляция вовсе не обязана сопровождаться значительными затратами времени выполнения. Как показано в примере, тело интерфейсной функции может быть написано внутри объявления класса; в этом случае функция является *подставляемой (встраиваемой, inline) функцией*, т.е. не используется механизм вызова подпрограммы и возврата из нее (см. гл. 7). Вместо этого код тела подпрограммы вставляется непосредственно внутрь последовательности кода в точке вызова. Поскольку при подстановке функции мы расплачиваемся пространством за время, подпрограммы должны быть очень маленькими (не более двух или трех команд). Другой фактор, который следует рассмотреть перед подстановкой подпрограммы, это то, что она вводит дополнительные условия для компиляции. Если вы изменяете подставляемую подпрограмму, все клиенты должна быть перекомпилированы.

14.3. Наследование

В разделе 4.6 мы показали, как в языке Ada один тип может быть получен из другого так, что производный тип получает копии значений и операций, которые были определены для порождающего типа. Задав порождающий тип:

```
package Airplane_Package is
   type Airplane_Data is
      record
         ID:String(1..80);                                    Ada
        Speed: Integer range 0.. 1000;
        Altitude: Integer range 0..100;
      end record;
    procedure New_Airplane(Data: in Airplane_Data: I; out Integer);
     procedure Get_Airplane(l: in Integer; Data: out Airplane_Data);
end Airplane_Package;
```

производный тип можно объявить в другом пакете:

```
type New_Airplane_Data is
new Airplane_Package.Airplane_Data;
```
Ada

Можно объявлять новые подпрограммы, которые выполняют операции на производном типе, и заменять подпрограммы родительского типа новыми:

```
procedure Display_Airplane(Data: in New_Airplane_Data);
                -- Дополнительная подпрограмма
procedure Get_Airplane(Data: in New_Airplane_Data; I: out Integer);
                -- Замененная подпрограмма
                -- Подпрограмма New_Airplane скопирована из Airplane_Data
```
Ada

Производные типы образуют семейство типов, и значение любого типа из семейства может быть преобразовано в значение другого типа из этого семейства:

```
A1: Airplane_Data;
A2: New_Airplane_Data := New_Airplane_Data(A1);
A3: Airplane_Data := Airplane_Data(A2);
```
Ada

Более того, можно даже получить производный тип от приватного типа, хотя, конечно, все подпрограммы для производного типа должны быть определены в терминах общих подпрограмм родительского типа.

Проблема, связанная с производными типами в языке Ada, заключается в том, что могут быть расширены только *операции*, но не компоненты данных, которые образуют тип. Например, предположим, что система управления воздушным движением должна измениться так, чтобы для сверхзвукового самолета в дополнение к существующим данным хранилось число Маха. Одна из возможностей состоит в том, чтобы просто включить дополнительное поле в существующую запись. Это приемлемо, если изменение делается при первоначальной разработке программы. Однако, если система уже была протестирована и установлена у заказчика, лучше будет найти решение, которое не требует перекомпиляции и проверки всего существующего исходного кода.

В таком случае лучше использовать *наследование (inheritance)*, которое является способом расширения существующего типа, не только путем добавления и изменения операции, но и добавления данных к типу. В языке C++ это реализовано через порождение одного класса из другого:

```
class SST_Data: public Airplane_Data {
private:
    float mach;
public:
    float get_mach() const {return mach;};
     void set_mach(float m) {mach = m;};
};
```
C++

Производный класс SST_Data получен из существующего класса Airplane_Data. Это означает, что каждый элемент данных и каждая подпрограмма, которые определены для *базового класса (base class)*, доступны и в производном классе. Кроме того, каждое значение производного класса SST_Data будет иметь дополнительный компонент данных mach, и есть две новые подпрограммы, которые могут применяться к значениям производного типа.

Производный класс — это обычный класс в том смысле, что могут быть объявлены экземпляры и вызваны подпрограммы:

C++

```
SST_Data s;
s.set_speed(1400);          //Унаследованная подпрограмма
s.set_mach(2.4);            // Новая подпрограмма
```

Подпрограмма, вызванная для set_mach, — это подпрограмма, которая объявлена внутри класса SST_ Data, а подпрограмма, вызванная для set_speed, — это подпрограмма, которая унаследована от базового класса. Обратите внимание, что производный класс может быть откомпилирован и скомпонован без изменения и перекомпиляции базового класса; таким образом, расширение на существующий код воздействовать не должно.

14.4. Динамический полиморфизм в языке C++

Когда один класс порожден из другого класса, вы можете *замещать (override)* унаследованные подпрограммы в производном классе, переопределяя их:

```
class SST_Data: public Airplane_Data {
public:
     int get_spaed() const;          // Заместить
     void set_speed(int):            // Заместить
};
```

Если задан вызов:

```
obj.set_speed(100);
```

то решение, какую именно из подпрограмм вызвать — подпрограмму, унаследованную из Airplane_Data, или новую в SST_ Data, — принимается во время компиляции на основе класса объекта obj.Это называется *статическим связыванием (static binding),* или *ранним связыванием (early binding),* так как решение принимается до выполнения программы, и при выполнении всегда вызывается одна и та же подпрограмма.

Однако вся суть наследования состоит в том, чтобы создать группу классов с аналогичными свойствами, и резонно ожидать, что должна иметься возможность присвоить переменной значение, принадлежащее любому из этих классов. Что должно произойти, когда вызывается подпрограмма для такой переменной? Решение, какую подпрограмму вызывать, должно быть принято во время выполнения, потому что значение, содержащееся в переменной, до этого неизвестно; фактически, переменная может содержать значения разных классов в разное время выполнения программы. Термины, используемые для обозначения способности выбирать подпрограммы во время выполнения, — *динамический полиморфизм, динамическое связывание, позднее связывание* и *диспетчеризация во время выполнения (dynamic polymorphism, dynamic binding, late binding и run-time dispatching).*

В языке C++ используются *виртуальные функции (virtual functions)* для обозначения тех подпрограмм, для которых выполняется динамическое связывание:

```
class Airplane_Data {
private:
…
public:
    virtual int get_speed() const;
     virtual void set_speed(int);
….
};
```

Подпрограмма в производном классе с тем же самым именем и сигнатурой параметров, что и виртуальная подпрограмма в порождающем классе, также считается виртуальной. Повторять спецификатор virtual необязательно, но это лучше сделать для ясности:

```
class SST_Data : public Airplane_Data {
private:
        float mach;
public:
          float get_mach() const;                  // Новая подпрограмма
          void set_mach(float m);                  // Новая подпрограмма
          virtual int get_speed() const;           // Заместить виртуальную подпрограмму
          virtual void set_speed(int);             // Заместить виртуальную подпрограмму
…
};
```

Рассмотрим теперь процедуру update со ссылочным параметром на базовый класс:

```
void update(Airplane_Data & d, int spd, int alt)
}
d.set_speed(spd);                          // На какой тип указывает d??
d.set altitude(alt);                       //На какой тип указывает d??
}

Airplane_Data a;
SST_Data s;

void proc()
{
          update(a, 500, 5000);                  // Вызвать с AirplaneJData
          update(s, 800,6000);                   // Вызвать с SST_Data
}
```

Идея производных классов состоит в том, что производное значение является базовым значением (возможно, с дополнительными полями), поэтому update может вызываться с параметром s производного класса SST_Data. При компиляции update компилятор не может знать, на что указывает d: на значение Airplane_Data или на SST_Data. Поэтому он не может однозначно скомпилировать вызов set_speed, поскольку эта подпрограмма по-разному определена в двух классах. Следовательно, компилятор должен сгенерировать код для переключения *(диспетчеризации)* вызова на правильную подпрограмму во время выполнения в зависимости от того, на что указывает d. В первом вызове proc указатель d указывает на Airplane_Data, и вызов будет диспет-черизован на подпрограмму, определенную в классе Airplane_Data, тогда как второй — на подпрограмму, определенную в SST_ Data.

Позвольте нам подчеркнуть преимущества динамического полиморфизма: вы можете писать большие блоки программы полностью в общем виде, используя вызовы виртуальных подпрограмм. Специализация обработки конкретного класса в семействе производных классов делается только во время выполнения за счет диспетчеризации виртуальных подпрограмм. Кроме того если вам когда-либо понадобится добавить производные классы в семействе не нужно будет изменять или перекомпилировать ни один из существующих кодов, потому что любое изменение в существующей программе ограничено исключительно новыми реализациями виртуальных подпрограмм. Например если мы порождаем еще один класс:

```
class Space_Plane_Data : public SST_Data {
      virtual void set_speed(int);                    // Заместить виртуальную подпрограмм
private:
       int reentry_speed;
};

Space_Plane_Data sp;
update(sp, 2000,30000);
```

файл, содержащий определение для update, не нужно перекомпилировать, даже если а) новая подпрограмма заместила set_speed и б) значение формального параметра d в update содержит дополнительное поле reentry_speed.

Когда используется динамический полиморфизм?

Давайте объявим базовый класс с виртуальной подпрограммой и обычной невиртуальной подпрограммой и породим класс, который добавляет дополнительное поле и дает новые объявления для обеих подпрограмм:

```
class Base_Class {
private:
    int Base_Field;
public:
     virtual void virtual_proc();
     void ordinary_proc();
};
class Derived_Class : public Base_Class {
private:
      int Derived_Field;
public:
     virtual void virtual_proc();
      void ordnary_proc(); };
```

Затем объявим экземпляры классов в качестве переменных. Присваивание значения производного класса переменной из базового класса разрешено:

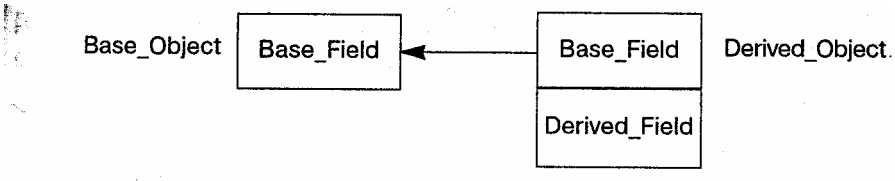

Рис. 14.3. Прямое присваивание производного объекта.

```
Base_Class          Base_0bject;
Derived_Class       Derived_Object;
if (...) Base_0bject = Derived_Object;
```

потому что производный объект является базовым объектом (плюс дополнительная информация), и при присваивании дополнительная информация может игнорироваться (см. рис. 14.3).

Более того, вызов подпрограммы (виртуальной или не виртуальной) однозначный, и компилятор может использовать статическое связывание:

```
Base_0bject .virtual_proc();
Base_Object.ordinary_proc();
Derived_0bject.virtual_proc();
Derived_0bject.ordinary_proc();
```

Предположим, однако, что используется косвенность, и указатель на производный класс присвоен указателю на базовый класс:

```
Base_Class*          Base_Ptr = new Base_Class;
Derived_Class*       Derived_Ptr = new Derived_Class;
if (...) Base_Ptr = Derived_Ptr;
```

В этом случае семантика другая, так как базовый указатель ссылается на *полный* производный объект без каких-либо усечений (см. рис. 14.4). При реализации не возникает никаких проблем, потому что мы принимаем, что все указатели представляются одинаково независимо от указуемого типа.

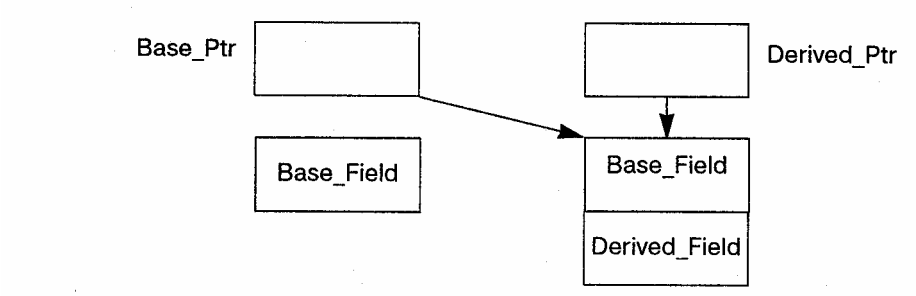

Рис. 14.4. Косвенное присваивание производного объекта.

Важно обратить внимание на то, что после присваивания указателя *компилятор* больше не имеет никакой информации относительно типа указуемого объекта. Таким образом, у него нет возможности привязать вызов

```
Base_Ptr- >virtual_proc();
```

к правильной подпрограмме, и следует выполнить динамическую диспетчеризацию. Аналогичная ситуация возникает, когда используется ссылочный параметр, как было показано выше.

Эта ситуация может внести путаницу, так как программисты обычно не делают различия между переменной и указуемым объектом. После следующих операторов:

```
inti1 = 1;
int i2 = 2;
int *p1 = &i1;          // p1 ссылается на i1
int *p2 = &i2;          // p2 ссылается на i2
p1 = p2;                // p1 также ссылается на i2
i1 = i2;                // i1 имеет то же самое значение, что и i2
```

вы ожидаете, что i1 == i2 и *p1 ==*p2; это, конечно, правильно, пока типы в точности совпадают, но это неверно для присваивания производного класса базовому классу из-за усечения. При использовании наследования вы должны помнить, что указуемый объект может иметь тип, отличный от типа указуемого объекта в объявлении указателя.

Есть одна западня в семантике динамического полиморфизма языка C++: если вы посмотрите внимательно, то заметите, что обсуждение касалось диспетчеризации, относящейся к замещенной *виртуальной* подпрограмме. Но в классе могут также быть и обычные подпрограммы, которые замещаются:

```
Base_Ptr = Derived_Ptr;
Base_Ptr->virtual_proc();       // Диспетчеризуется по указанному типу
Base_Ptr->ordinary_proc();      // Статическое связывание с базовым типом!!
```

Существует различие в семантике между двумя вызовами: вызов виртуальной подпрограммы диспетчеризуется во время выполнения в соответствии с *типом указуемого объекта,* в данном случае Derived_Class; вызов обычной подпрограммы связывается статически во время компиляции в соответствии с *типом указателя,* ъ данном случае Base_Class. Это различие весьма существенно, потому что изменение, которое состоит в замене невиртуальной подпрограммы на виртуальную подпрограмму или обратно, может вызвать ошибки во всем семействе классов, полученных из базового.

Динамическая диспетчеризация в языке C++ рассчитана на вызовы виртуальных подпрограмм, осуществляемые через указатель или ссылку.

Реализация

Ранее мы отмечали, что если подпрограмма не найдена в производном классе, то поиск делается в предшествующих классах, пока не будет найдено определение подпрограммы. В случае статического связывания поиск можно делать во время компиляции: компилятор просматривает базовый класс производного класса, затем его базовый класс, и так далее, пока не будет найдено соответствующее связывание подпрограммы. Затем для этой подпрограммы может компилироваться обычный вызов процедуры.

Если используются виртуальные подпрограммы, ситуация усложняется, потому что фактическая подпрограмма, которая должна быть вызвана, не известна до времени выполнения. Обратите внимание, что, если виртуальная подпрограмма вызывается с объектом конкретного типа, в противоположность ссылке или указателю, то все еще может использоваться статическое связывание. С другой стороны, решение, какую именно подпрограмму следует вызвать, основано на 1) имени подпрограммы и 2) классе объекта. Но первое известно во время компиляции, поэтому нам остается только смоделировать case-оператор по классам.

Обычно реализация выглядит немного иначе; для каждого класса с виртуальными подпрограммами поддерживается таблица диспетчеризации (см. рис. 14.5). Каждое значение класса должно «иметь при себе» свой *индекс* для входа в таблицу диспетчеризации для порождающего семейства, в котором оно определено. Элементы таблицы диспетчеризации являются указателями на таблицы переходов; в каждой таблице переходов содержатся адреса входов в виртуальные подпрограммы. Обратите внимание, что два элемента таблицы переходов могут указывать на одну и ту же процедуру; это произойдет, когда класс не замещает виртуальную подпрограмму. На рисунке cls3 произведен из

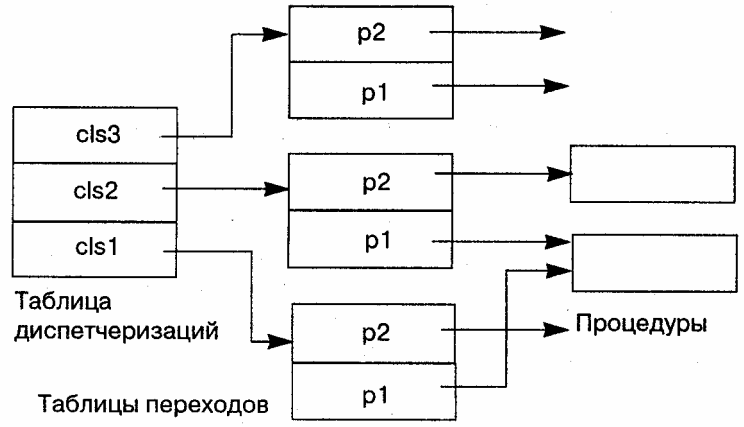

Рис. 14.5. Реализация динамического полиморфизма.

cls2, который в свою очередь произведен из базового класса cls1. Здесь cls2 заместил p2, но не p1, в то время как cls3 заместил обе подпрограммы.
Когда встречается вызов диспетчеризуемой подпрограммы ptr->p1(), выполняется код наподобие приведенного ниже, где мы подразумеваем, что неявный индекс — это первое поле указуемого объекта:

```
load    RO.ptr          Получить адрес объекта
load    R1 ,(RO)        Получить индекс указуемого объекта
load    R2,&dispatch    Получить адрес таблицы отправлений
add     R2.R1           Вычислить адрес таблицы переходов
load    R3,(R2)         Получить адрес таблицы переходов
load    R4,p1(R3)       Получить адрес процедуры
call    (R4)            Вызвать процедуру, адрес которой находится в R4
```

Даже без последующей оптимизации затраты на время выполнения относительно малы, и, что более важно, *фиксированы*, поэтому в большинстве приложений нет необходимости воздерживаться от использования динамического полиморфизма. Но все же издержки существуют и применять динамический полиморфизм следует только после тщательного анализа. Лучше избегать обеих крайностей: и чрезмерного использования динамического полиморфизма только потому, что это «хорошая идея», и отказа от него, потому что это «неэффективно».

Обратите внимание, что фиксированные затраты получаются благодаря тому, что динамический полиморфизм ограничен фиксированным набором классов, порожденных из базового класса (поэтому может использоваться таблица диспетчеризации фиксированного размера), и фиксированным набором виртуальных функций, которые могут быть переопределены (поэтому размер каждой таблицы переходов также фиксирован). Значительным достижением языка C++ была демонстрация того, что динамический полиморфизм может быть реализован без неограниченного поиска во время выполнения.

14.5. Объектно-ориентированное программирование на языке Ada 95

В языке Ada 83 наличие пакетной конструкции обеспечивает полную поддержку инкапсуляции, а наличие производных типов частично обеспечивает наследование. Полного наследования нет, потому что, когда вы производите новый тип, то можете добавлять только новые операции, но не новые компоненты данных. Кроме того, единственный полиморфизм — это статический полиморфизм вариантных записей. В языке Ada 95 поддерживается полное наследование за счет того, что программисту дается возможность *расширить* запись производного типа. Чтобы обозначить, что родительский тип записи пригоден для наследования, его нужно объявить как *теговый (tagged)* тип записи:

```
package Airplane_Package is
    type Airplane_Data is tagged
        record
            ID:String(1..80);
            Speed: Integer range 0..1000;
            Altitude: Integer range 0..100;
        end record;
end Airplane_Package;
```

Этот тег аналогичен тегу в языке Pascal и дискриминанту в вариантных записях языка Ada, где он используется для того, чтобы различать разные типы, производные друг из друга. В отличие от этих конструкций, тег теговой записи неявный, и программист не должен явно к нему обращаться. Заглядывая вперед, скажем, что этот неявный тег будет использоваться, чтобы диспетчери-зовать вызовы подпрограмм для динамического полиморфизма.
Чтобы создать абстрактный тип данных, тип должен быть объявлен как приватный и полное объявление типа дано в закрытой части:

```
package Airplane_Package is
           type Airplane_Data is tagged private;
           procedure Set_ID(A: in out Airplane_Data; S: in String);
           function Get_ID(A: Airplane_Data) return String;
           procedure Set_Speed(A: in out Airplane_Data; I: in Integer);
           function Get_Speed(A: Airplane_Data) return Integer;
            procedure Set_Altitude(A: in out Airplane_Data; I: in Integer);
            function Get_Altitude(A: Airplane_Data) return Integer;
private
      type Airplane_Data is tagged
        record
           ID:String(1..80);
           Speed: Integer range 0..1000;
           Altitude: Integer range 0.. 100;
         end record;
end Airplane_Package;
```

Подпрограммы, определенные *внутри* спецификации пакета, содержащей объявление тегового типа (наряду со стандартными операциями на типе), называются *примитивными операциями*, или *операциями-примитивами (primitive operations)* и являются подпрограммами, которые наследуются. Наследование выполняется за счет *расширения (extending)* тегового типа:

```
with Airplane_Package; use Airplane_Package;
package SST_Package is
      type SST_Data is new Airplane_Data with
        record
          Mach: Float;
        end record;
      procedure Set_Speed(A: in out SST_Data; I: iri Integer);
      function Get_Speed(A: SST_Data) return Integer;
end SST_Package;
```

Значения этого производного типа являются копиями значений родительского типа Airplane_Data вместе с (with) дополнительным полем записи Mach. Операции, определенные для этого типа, являются копиями элементарных подпрограмм; эти операции могут быть замещены. Конечно, для производного типа могут быть объявлены другие самостоятельные подпрограммы.

В языке Ada нет специального синтаксиса для вызова подпрограмм-примитивов:

```
A: Airplane_Data;
Set_Speed(A, 100);
```

С точки зрения синтаксиса объект А — это обычный параметр; И по его типу компилятор может решить, какую именно подпрограмму вызвать. Параметр называется *управляющим,* Потому что он управляет тем, какую подпрограмму выбрать. *Управляющий параметр* не обязан быть первым параметром, и их может быть несколько (при условии, что все они того же типа). Сравните это с языком C++, который использует специальный синтаксис, чтобы вы-звать подпрограмму, объявленную в классе:

```
Airplane_Data a;
a.set_speed(100);
```

C++

Объект a является *отличимым получателем (distinguished receiver)* сообщения set_speed. Отличимый получатель является неявным параметром, в данном случае обозначающим, что скорость (speed) будет установлена (set) для объекта a.

Динамический полиморфизм

Перед обсуждением динамического полиморфизма в языке Ada 95 мы должны коснуться различий в терминологии языка Ada и других объектно-ориентированных языков.

В языке C++ термин *класс* обозначает тип данных, который используется для создания экземпляров объектов этого типа. Язык Ada 95 продолжает использовать термины *типы* и *объекты* даже для теговых типов и объектов, которые известны в других языках как классы и экземпляры. Слово *класс* ис-| пользуется для обозначения набора всех типов, которые порождаются от об-|щего предка, в языке C++ мы их назвали семейством классов. Нижеследующее обсуждение лучше всего провести в правильной терминологии языка Ada 95; будьте внимательны и не перепутайте новое применение слова *класс* с его использованием в языке C++.

С каждым теговым типом T связан тип, который обозначается как T'Class и называется *типом класса (class-wide type)".* T'Class *покрывает (covered)* все типы, производные от T. Тип класса — это неограниченный тип, и объявить объект этого типа, не задав ограничений, нельзя, подобно объявлению неограниченного массива:

```
type Vector is array(lnteger range <>) of Float;
V1: Vector;                                    -- Запрещено, нет ограничений
type Airplane_Data is tagged record . . . end record;
A1: Airplane_Data'Class:                       -- Запрещено, нет ограничений
```

Объект типа класса может быть объявлен, если задать начальное значение:

```
V2: Vector := (1 ..20=>0.0);                   -- Правильно, ограничен
X2: Airplane_Data;                             -- Правильно, конкретный тип
X3: SST_Data;                                  -- Правильно, конкретный тип
A2: Airplane_Data'Class := X2;                 -- Правильно, ограничен
A3: Airplane_Data'Class := X3;                 --Правильно, ограничен
```

Как и в случае массива, коль скоро CW-объект ограничен, его ограничения изменить нельзя. CW-тип можно использовать в декларации локальных переменных подпрограммы, которая получает параметр CW-типа. Здесь снова полная аналогия с массивами:

```
procedure P(S: String; C: in Airplane_Data'Class) is
    Local_String: String := S;
    Local_Airplane: Airplane_Data'Class := C;
Begin
…
end P;
```

Динамический полиморфизм имеет место, когда *фактический параметр имеет* тип класса, в то время как *формальный параметр* — конкретного типа, принадлежащего классу:

```
with Airplane_Package; use Airplane_Package;
with SST_Package; use SST_Package;
procedure Main is
    procedure Proc(C: in out Airplane_Data'Class; I: in Integer) is
    begin
       Set_Speed(C, I);                          -- Какого типа C ??
    end Proc;
    A: Airplane_Data;
    S: SST_Data;
begin       -- Main
  Proc(A, 500);                                  -- Вызвать с Airplane_Data
   Proc(S, 1000);                                -- Вызвать с SST_Data end Main:
```

Фактический параметр C в вызове Set_Speed имеет тип класса, но имеются две версии Set_Speed с *формальным* параметром либо родительского типа, либо производного типа. Во время выполнения тип C будет изменяться от вызова к вызову, поэтому динамическая диспетчеризация необходима, чтобы снять неоднозначность вызова.

Рисунок 14.6 поможет вам понять роль формальных и фактических параметров в диспетчеризации. Вызов Set_Speed вверху рисунка делается с *фактическим* параметром типа класса. Это означает, что только при вызове подпрограммы мы знаем, имеет ли фактический параметр тип Airplane_Data или SST_Data. Однако каждое обтъявление процедуры, показанное внизу рисунка, имеет формальный параметр конкретного типа. Как показано стрелками, вызов должен быть отправлен в соответствии с типом фактического параметра.

Обратите внимание, что диспетчеризация выполняется только в случае необходимости; если компилятор может разрешить вызов статически, он так и сделает. Следующие вызовы не нуждаются ни в какой диспетчеризации, потому что вызов делается с *фактическим* параметром конкретного типа, а не типа класса:

```
Set_Speed(A, 500);
Set_Speed(S, 1000);
```

Точно так же, если *формальный* параметр имеет тип класса, то никакая диспетчеризация не нужна. Вызовы Proc — это вызовы отдельной однозначной про-

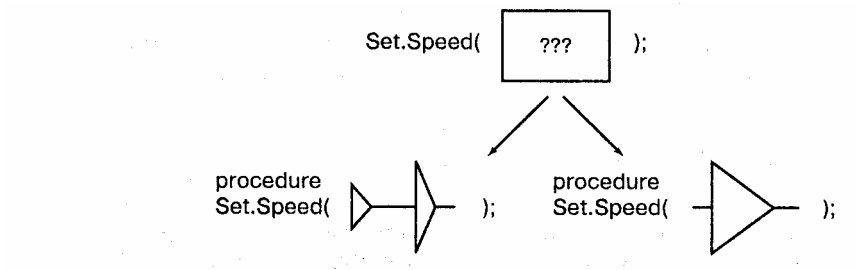

Рис. 14.6. Динамическая диспетчеризация в языке Ada 95.

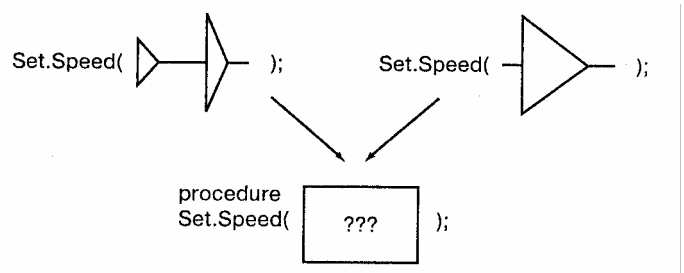

Рис. 14.7. Формальный CW-параметр.

цедуры; формальный параметр имеет тип класса, который соответствует фактическому параметру любого типа, относящегося к классу. Что касается рис. 14.7, то, если бы объявление Set_Speed было задано как:

```
procedure Set_Speed(A: in out Airplane'Class: I: in Integer);
```

то любой фактический параметр класса «вписался» бы в формальный параметр класса. Никакая диспетчеризация не нужна, потому что каждый раз вызывается одна и та же подпрограмма.

При ссылочном доступе указуемый объект так же может иметь CW-тип. Указатель при этом может указывать на любой объект, тип которого покрывается CW-типом, и диспетчеризация осуществляется просто раскрытием указателя:

```
type Class_Ptr is access Airplane_Data'Class;
Ptr: Class_Ptr := new Airplane_Data;
if (...) then Ptr := new SST_Data; end if;
Set_Speed(Ptr.all);                          -- На какой именно тип указывает Ptr??
```

Динамический полиморфизм в языке Ada 95 имеет место, когда фактический параметр относится к CW-типу, а формальный параметр относится к конкретному типу.

Реализации диспетчеризации во время выполнения в языках Ada 95 и C++ похожи, тогда *как условия* для диспетчеризации совершенно разные:

• В C++ подпрограмма должна быть объявлена виртуальной, чтобы можно было выполнить диспетчеризацию. Все косвенные вызовы виртуальной подпрограммы диспетчеризуются.

• В языке Ada 95 любая унаследованная подпрограмма может быть замещена и неявно становится диспетчеризуемой. Диспетчеризация выполняется только в случае необходимости, если этого требует конкретный вызов.

Основное преимущество подхода, принятого в языке Ada, состоит в том, что не нужно заранее определять, должен ли использоваться динамический полиморфизм. Это означает, что не существует различий в семантике между вызовом виртуальной и невиртуальной подпрограммы. Предположим, что Airplane_Data был определен как теговый, но никакие порождения сделаны не были. В этом случае вся система построена так, что в ней все вызовы разрешены статически. Позже, если будут объявлены производные типы, они смогут использовать диспетчеризацию без изменения или перекомпиляции существующего кода.

14.6. Упражнения

1. Метод разработки программного обеспечения, называемый *нисходящим* программированием, пропагандирует написание программы в терминах операций высокого уровня абстракции и последующей постепенной детализации операций, пока не будет достигнут уровень операторов языка программирования. Сравните этот метод с объектно-ориентированным программированием.

2. Объявили бы вы Aircraft_Data абстрактным типом данных или сделали поля класса открытыми?

3. Проверьте, что можно наследовать из класса в языке C++ или из тегового пакета в языке Ada 95 без перекомпиляции существующего кода.

4. Опишите неоднородную очередь на языке Ada 95: объявите теговый тип Item, определите очередь в терминах Item, а затем породите из Item производные типы — булев, целочисленный и символьный.

5. Опишите неоднородную очередь на языке C++.

6. Проверьте, что в языке C++ диспетчеризация имеет место для ссылочного, но не для обычного параметра.

7. В языке Ada 95 теговый тип может быть расширен приватными добавлениями:

```
with Airplane_Package; use Airplane_Package;
package SST_Package is
      type SST_Data is new Airplane_Data with private;
      procedure Set_Speed(A: in out SST_Data; I: in Integer);
      function Get_Speed(A: SST_Data) return Integer;
private
…
end SST_Package;
```

Каковы преимущества и недостатки такого расширения?

8. Изучите машинные команды, сгенерированные компилятором Ada 95 или C++ для динамического полиморфизма.

Глава 15

Еще об объектно-ориентированном программировании

В этой главе мы рассмотрим еще несколько конструкций, которые существуют в объектно-ориентированных языках. Это не просто дополнительные удобства — это существенные конструкции, которые необходимо освоить, если вы хотите стать компетентными в объектно-ориентированных методах программирования. Данный обзор не является исчерпывающим; детали можно уточнить в учебниках по языкам программирования. Глава разделена на шесть разделов:

1. Структурированные классы.
• Абстрактные классы используются для создания абстрактного интерфейса, который можно реализовать с помощью одного или нескольких наследуемых классов.
• Родовые подпрограммы (Ada) и шаблоны (C++) можно комбинировать с наследованием для параметризации классов другими классами.
• Множественное наследование: класс может быть производным от двух или нескольких родительских классов и наследовать данные и операции каждого из них.

2. Доступ к приватным компонентам: Являются компоненты в закрытой части пакета или класса всегда приватными, или их можно экспортировать производным классам или клиентам?

3. Данные класса. В этом разделе обсуждаются создание и использование компонентов данных в классе.

4. Eiffel. Язык Eiffel был разработан для поддержки ООП как единственного метода структурирования программ; поучительно сравнить конструкции языка Eiffel с конструкциями языков Ada 95 и C++, где поддержка ООП была добавлена к уже существующим языкам.

5. Проектные соображения. Каковы компромиссы между использованием класса и наследованием из класса? Для чего может использоваться наследование? Каковы взаимоотношения между перегрузкой и замещением?

6. **В заключение** приводится сводка методов динамического полиморфизма.

15.1. Структурированные классы

Абстрактные классы

Когда класс порождается из базового класса, предполагается, что базовый класс содержит большую часть требуемых данных и операций, тогда как производный класс всего лишь добавляет дополнительные данные, а также добавляет или изменяет некоторые операции. Во многих проектах лучше рассматривать базовый класс как некий каркас, определяющий общие операции для всего семейства производных классов. Например, семейство классов операций ввода/вывода или графики может определять такие общие операции, как get и display, которые будут определены для каждого производного класса. И Ada 95, и C++ поддерживают такие абстрактные классы.

Мы продемонстрируем абстрактные классы, описывая несколько реализаций одной и той же абстракции; абстрактный класс будет определять структуру данных Set, и производные классы — реализовывать множества двумя различными способами. В языке Ada 95 слово abstract обозначает абстрактный тип и абстрактные подпрограммы, связанные с этим типом:

```
package Set_Package is
      type Set is abstract tagged null record;
      function Union(S1, S2: Set) return Set is abstract;
      function Intersection(S1, S2: Set) return Set is abstract;
end Set_Package;
```

Ada

Вы не можете объявить объект абстрактного типа и не можете вызвать абстрактную подпрограмму. Тип служит только каркасом для порождения конкретных типов, а подпрограммы должны замещаться конкретными подпрограммами.

Сначала мы рассмотрим производный тип, в котором множество представлено булевым массивом:

```
with Set_Package;
package Bit_Set_Package is
     type Set is new Set_Package.Set with private;
     function Union(S1, S2: Set) return Set;
     function lntersection(S1, S2: Set) return Set;
private
    type Bit_Array is array(1..100) of Boolean;
    type Set is new Set_Package.Set with
       record
          Data: Bit_Array;
       end record;
end Bit_Set_Package;
```

Ada

Конечно, необходимо тело пакета, чтобы реализовать операции.
Производный тип — это конкретный тип с конкретными компонентами данных и операциями, и он может использоваться как любой другой тип:

```
with Bit_Set_Package; use Bit_Set_Package;
procedure Main is
     S1.S2, S3: Set;
begin
      S1 := Union(S2, S3);
end Main;
```

Ada

Предположим теперь, что в другой части программы требуется другая реализация множеств, которая использует связанные списки вместо массивов. Вы можете породить

дополнительный конкретный тип из абстрактного типа и использовать его вместо или в дополнение к предыдущей реализации:

```
with Set_Package;
package Linked_Set_Package is
     type Set is new Set_Package.Set with private;
     function Union(S1, S2: Set) return Set;
     function lntersection(S1, S2: Set) return Set;
private
     type Node;
     type Pointer is access Node;
      type Set is new Set_Package.Set with
         record
               Head: Pointer;
        end record;
end Linked_Set_Package;
```

Ada

Новая реализация может использоваться другим модулем; фактически, вы можете изменить реализацию, используемую в существующих модулях, просто заменяя контекстные указания:

```
Ada with Linked_Set_Package; use Linked_Set_Package;
procedure Main is
      S1.S2, S3: Set;
begin
      S1 := Union(S2, S3);
end Main;
```

Ada

В C++ абстрактный класс создается с помощью объявления *чистой* виртуальной функции, обозначенной «начальным значением» 0 для функции.

Абстрактный класс для множеств в языке C++ выглядит следующим образом:

```
class Set {
public:
     virtual void Union(Set&, Set&) = 0;
     virtual void lntersection(Set&, Set&) = 0;
  };
```

C++

У абстрактных классов не бывает экземпляров; абстрактный класс может только быть базовым для производных классов:

```
class Bit_Set: public Set {
public:
    virtual void Union(Set&, Set&);
    virtual void lntersection(Set&, Set&);
private:
     int data[100];
};

class Linked_Set: public Set {
public:
    virtual void Union(Set&, Set&);
    virtual void lntersection(Set&, Set&);
private:
     int data;
     Set *next;
```

C++

```
};
```

Конкретные производные классы можно использовать как любой другой класс: __

```
void proc()
{
Bit_Setb1,b2, b3;
Linked_Set 11,12,l3;

b1.Union(b2,b3);
H.Union(l2,I3);
}
```

C++

Обратите внимание на разницу в синтаксисе двух языков, которая вызвана разными подходами к ООП. В языке Ada 95 определяется обычная функция, которая получает два множества и возвращает третье. В языке C++ одно из множеств — отличимый получатель сообщения. Для

```
b1.Union(b2,b3);
```

подразумевается, что экземпляр b1, отличимый получатель операции Union, получит результат операции от двух параметров — Ь2 и b3 — и использует его, • чтобы заменить текущее значение внутренних данных.

Возможно, вы предпочтете перегрузить предопределенные операции, например «+» и «*», вместо того чтобы использовать имена Union и Intersection. Это можно сделать как в C++, так и в Ada 95.

Все реализации абстрактного класса покрываются типом класса (CW-типом) Set'Class. Величины абстрактного CW-типа будут диспетчеризованы к правильному конкретному типу, т. е. к правильной реализации. Таким образом, абстрактные типы и операции дают возможность программисту писать программное обеспечение, не зависящее от реализации.

Родовые возможности

В разделе 10.3 мы обсуждали родовые подпрограммы в языке Ada, которые позволяют программисту создавать шаблоны подпрограмм и затем конкретизировать их для различных типов. Родовые возможности чаще всего находят приложение в пакетах Ada; например, пакет работы со списком может быть родовым в отношении типа элементов списка. Кроме того, он может быть родовым в отношении функций, сравнивающих элементы, с тем чтобы элементы списка можно было сортировать:

```
generic
    type Item is private;
    with function "<"(X, Y: in Item) return Boolean;
package List_Package is
    type List is private;
    procedure Put(l: in Item; L: in out List);
    procedure Get(l: out Item; L: in out List);
private
    type List is array( 1.. 100) of Item;
end List_Package;
```

Ada

Этот пакет теперь может быть конкретизирован для любого типа элемента:

```
package Integer_list is new List_Package(lnteger, Integer."<");
```
Ada

Конкретизация создает новый тип, и можно объявлять и использовать объекты этого типа:

```
lnt_List_1, lnt_List_2: lnteger_List.List;
lnteger_List.Put(42, lnt_List_1 );
lnteger_List.Put(59, lnt_List_2);
```

В языке Ada есть богатый набор нотаций для написания родовых формальных параметров, которые используются в модели контракта, чтобы ограничить фактические параметры некоторыми классами типов, такими как дискретные типы или типы с плавающей точкой. В языке Ada 95 эти средства обобщены до возможности специфицировать в родовом формальном параметре классы типов, задаваемые программистом:

```
with Set_Package;
generic
    type Set_Class is new Set_Package.Set; package Set_IO is
…
end Set_IO;
```
Ada

Эта спецификация означает, что родовой пакет может быть конкретизирован с любым типом, производным от тегового типа Set, такого как Bit_Set и Linked_Set. Все операции из Set, такие как Union, могут использоваться внутри родового пакета, потому что из модели контракта мы знаем, что любая конкретизация будет с типом, производным от Set, и, следовательно, она наследует или замещает эти операции.

Шаблоны

В языке C++ можно определять шаблоны классов:

```
template <class ltem>
   class List {
        void put(const Item &);
};
```
Ada

Как только шаблон класса определен, вы можете определять объекты этого класса, задавая параметр шаблона:

```
List<int>lnt_List1;
     // lnt_List1 является экземпляром класса List с параметром int
```
C++

Так же как и язык Ada, C++ позволяет программисту для объектов-экземпляров класса задать свои программы (процесс называется *специализацией, specialization)* или воспользоваться по умолчанию подпрограммами, которые существуют для класса. Есть важное различие родовых пакетов Ada и шаблонов C++. В языке Ada конкретизация родового пакета, который определяет тип, даст вам конкретный *пакет,* содержащий конкретный тип. Чтобы получить объект, потребуется еще один шаг. В C++ конкретизация дает объект сразу, не определяя конкретного класса. Чтобы определить другой объект, нужно просто конкретизировать шаблон снова:

```
List<int>Int_List2;                    //Другой объект
```
C++

Компилятор и компоновщик отвечают за то, чтобы отследить пути всех конкретизации одного и того же типа и гарантировать, что код для операций шаблона класса не тиражируется для каждого объекта.

Следующее различие между языками состоит в том, что C++ не использует модель контракта, поэтому не исключено, что конкретизация вызовет ошибку компиляции в самом шаблоне (см. раздел 10.3).

Множественное наследование

Ранее обсуждалось порождение классов от одного базового класса, так что семейство классов образовывало дерево. При объектно-ориентированном проектировании, вероятно, класс будет иметь характеристики двух или нескольких существующих классов, и кажется допустимым порождать класс из нескольких базовых классов. Это называется *множественным наследованием (multiple inheritance).* На рисунке 15.1 показано, что Airplane (самолет) может

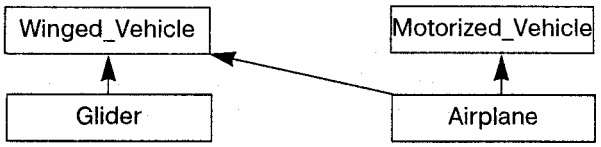

Рис. 15.1. Множественное наследование.

быть многократно порожден из Winged_Vehicle (летательный аппарат с крыльями) и Motorized_Vehicle (летательный аппарат с мотором), в то время как Winged_Vehicle также является (единственным) базовым классом для Glider (планер). Задав два класса:

```
class Winged_Vehicle {
public:
   void display(int);
protected:
     int Wing_Length;                    // Размах крыла
     int Weight;                         // Вес
};
class Motorized_Vehicle {
public:
     void display(int);
protected:
     int Power;                          // Мощность
      int Weight;                        // Вес
};
```

C++

можно породить класс с помощью множественного наследования:

```
class Airplane:
      public Winged_Vehicle, public Motorized_Vehicle {
public:
        void display_all();
};
```

C++

Чтобы использовать множественное наследование, необходимо решить, что делать с данными и операциями, такими как Weight и display, которые наследуются из нескольких базовых классов. В языке C++ неоднозначность, вызванная многократно определенными компонентами, должна быть явно разрешена с помощью операции уточнения области действия:

```
void Airplane: :display_all()
{
      Winged_Vehicle::display(Wing_Length);
       Winged_Vehicle::display(Winged_ Vehicle:: Weight);
       Motorized_ Vehicle:: display(Power);
       Motorized_ Vehicle:: display(Motorized_ Vehicle:: Weight);
};
```

C++

Это нельзя считать удачным решением, так как вся идея наследования в том, чтобы допускался прямой доступ к данным и операциям базы, если не требуется их модификации. Реализовать множественное наследование намного труднее, чем простое наследование, которое мы описали в разделе 14.4. Более подробно см. разделы с 10.1c по 10.1c упомянутого ранее справочного руководства по языку C++.

Значение множественного наследования в ООП является предметом для дискуссии. Некоторые языки программирования, такие как Eiffel, поддерживают использование множественного наследования, в то время как языки, подобные Ada 95 и Smalltalk, не имеют таких средств. При этом утверждается, что проблемы, которые можно решить с помощью множественного наследования, изящно решаются с использованием других средств языка. Например, выше мы отмечали, что родовые параметры теговых типов в языке Ada 95 можно использовать для создания новых абстракций, комбинируя уже существующие абстракции. Очевидно, что наличие возможности множественного наследования оказывает глубокое влияние на проектирование и программирование объектно-ориентированной системы. Таким образом, трудно говорить об объектно-ориентированном проекте, не зависящем от языка; даже на самых ранних стадиях проектирования вам следует ориентироваться на конкретный язык программирования.

5.2. Доступ к приватным компонентам

<<Друзья>> в языке C++

Внутри объявления класса в языке C++ можно включать объявление *«друже-ственных»* (friend) подпрограмм или классов, представляющих собой под-программы или классы, которые имеют полный доступ к приватным данным операциям класса:

```
class Airplane_Data {
private:
    int speed;
    friend void proc(const Airplane_Data &, int &);
    friend class CL;
};
```

Подпрограмма pгос и подпрограммы класса CL могут обращаться к приватным компонентам Airplane_Data:

```
void proc(const Airplane_Data & a, int & i)
{
    i = a.speed;                                    // Правильно, мы — друзья
}
```

Подпрограмма proc может затем передавать внутренние компоненты класса, используя ссылочные параметры, или указатели, как показано выше. Таким образом, «друг» выставил на всеобщее обозрение все секреты абстракции.

Мотив для предоставления такого доступа к приватным элементам взят из операционных систем, в которых были предусмотрены механизмы явного предоставления привилегий, называемых *возможностями* (capabilities). Это понятие меньше соответствует языкам программирования, потому что одна из целей ООП состоит в том, чтобы создавать закрытые, пригодные для повторного использования компоненты. Идея «друзей» проблематична с проектной точки зрения, поскольку предполагает, что *компонент* располагает знанием о том, кто им воспользуется, а это определенно несовместимо с идеей многократного использования компонентов, которые вы покупаете или заимствуете из других проектов. Другая серьезная проблема, связанная с конструкцией friend, состоит в слишком частом использовании ее для «заплат» в программе, вместо переосмысления абстракции. Чрезмерное употребление конструкции friend, очевидно, разрушит абстракции, которые были так тщательно разработаны.

Допустимо применение «друзей», когда абстракция составлена из двух самостоятельных элементов. В этом случае могут быть объявлены два класса, которые являются «друзьями» друг друга. Например, предположим, что классу Keyboard (клавиатура) необходим прямой доступ к классу Display (дисплей), чтобы воспроизвести эхо-символ; и наоборот, класс Display должен быть в состоянии поместить символ, полученный из интерфейса сенсорного экрана, во внутренний буфер класса Keyboard:

```
class Display {
private:
    void echo(char c);
    friend class Keyboard;                          // Разрешить классу Keyboard вызывать echo
};

class Keyboard {
private:
     void put_key(char c);
     friend class Display;                          // Разрешить классу Display вызывать put_key
};
```

Использование механизма friend позволяет избежать как создания неоправданно большого числа открытых (public) подпрограмм, так и объединения двух классов в один большой класс только потому, что они имеют одну-един-ственную общую операцию.

С помощью friend можно также решить проблему синтаксиса, связанную с тем фактом, что подпрограмма в классе C++ имеет отличимый получатель, такой как obj1 при вызове obj1.proc(obj2). Это привносит в подпрограммы асимметрию, в противном случае они были бы симметричны по параметрам. Стандартный пример — перегрузка арифметических операций. Предположим, что мы хотим перегрузить «+» для комплексных чисел и в то же время позволить операции неявно преобразовать параметр с плавающей точкой в комплексное значение:

```
complex operator + (float);
complex operator + (complex);
```

Рассмотрим выражение x + y, где одна из переменных (x или y) может быть с плавающей точкой, а другая комплексной. Первое объявление правильно для комплексного x и плавающего y, потому что x+y эквивалентно x.operator+(y), и, стало быть, будет

диспетчеризованно отличимому получателю комплексного типа. Однако второе объявление для x+y, где x имеет тип с плавающей точкой, приведет к попытке диспетчеризоваться к операции с плавающей точкой, но операция была объявлена в комплексном классе.

Решение состоит в том, чтобы объявить эти операции как «друзей» класса, а не как операции класса:

```
friend complex operator + (complex, complex);
friend complex operator + (complex, float);
friend complex operator + (float, complex);
```

Хотя эта конструкция популярна в языке C++, на самом деле существует лучшее решение, при котором не требуется friend.

Оператор «+=» можно определить как функцию-член (см. справочное руководство, стр. 249), а затем «+» можно определить как обычную функцию за пределами класса:

```
complex operator + (float left, complex right)
{
      complex result = complex(left);
      result + = right;                              // Результат является отличимым получателем
      return result;
}
```

Спецификаторы доступа в языке C++

Когда один класс порождается из другого, мы вправе спросить, имеет ли производный класс доступ к компонентам базового класса. В следующем примере database (база данных) объявлена как приватная, поэтому она недоступна в производном классе:

```
class Airplanes {
private:
     Airplane_Data database [100];
};
class Jets : public Airplanes {
      void process Jet(int);
};
      void Jets::process_jet(int i)
 {
      Airplane_Data d = database[i] ;                    // Ошибка, нет доступа!
};
```

Если объявлен экземпляр класса Jets, он будет содержать память для database, но этот компонент недоступен для любой подпрограммы в производном классе.

Есть три спецификатора доступа в языке C++:

• Общий (public) компонент доступен для любого пользователя класса.

• Защищенный (protected) компонент доступен внутри данного класса и внутри производного класса.

• Приватный компонент доступен только внутри класса.

В примере, если database просто защищенный, а не приватный член класса, к нему можно обращаться из производного класса Jets:

```
class Airplanes {
protected:
      Airplane_Data database[100];
};
class Jets : public Airplanes {
     void process_jet(int);
};
     void Jets::process_jet(int i)
{
     Airplane_Data d = database[i];                    // Правильно, в производном классе
};
```

Однако это не очень хорошая идея, потому что она ставит под удар абстракцию. Вероятно, было бы лучше даже для производного класса манипулировать унаследованными компонентами, используя общие или защищенные *подпрограммы*. Тогда, если внутреннее представление изменяется, нужно изменить только несколько подпрограмм.

Язык C++ допускает изменение доступности компонентов класса при объявлении производного класса. Обычно порождение бывает общим (public). Так было во всех наших примерах, и при этом сохранялась доступность, заданная в базовом класс. Однако вы также можете задать приватное порождение, тогда и общие, и защищенные компоненты становятся приватными:

```
class Airplanes {
protected:
     Airplane_Data database [100];
};
class Jets : private Airplanes {                              // Приватное порождение
         void process_jet(int);
};
         void Jets::process_jet(int i)
{
          Airplane_Data d = database[i];                      // Ошибка, нет доступа
};
```

Пакеты-дети в языке Ada

В языке Ada только тело пакета имеет доступ к приватным объявлениям. Это делает невозможным непосредственное совместное использование пакетами приватных объявлений так, как это можно делать в языке C++ с защищенными объявлениями. В языке Ada 95 для совместного использования приватных объявлений доставлено специальное средство структурирования, так называемые *пакеты-дети (child packages).* Здесь мы ограничим обсуждение пакетов-детей только для этой цели, хотя они чрезвычайно полезны в любой ситуации, когда вы хотите расширить существующий пакет без его изменения или пере-компиляции.

Зададим приватный тип Airplane_Data, определенный в пакете:

```
package Airptane_Package is
      type Airplane_Data is tagged private;
private
      type Airplane_Data is tagged
         record
            ID:String(1..80);
           Speed: Integer range 0.. 1000;
           Altitude: Integer 0.. 100;
```

```
        end record;
end Airplane_Package;
```

Этот тип может быть расширен в пакете-ребенке:

```
package Airplane_Package.SST_Package is
    type SST_Data is tagged private;
    procedure Set_Speed(A: in out SST_Data; I: in Integer);
 private
    type SST.Data is new Airplane_Data with
        record
            Mach: Float;
        end record;
end Airplane_Package.SST_Package;
```

Если задан пакет P1 и его ребенок P1 .P2, то P2 принадлежит области родителя P1, как если бы он был объявлен сразу после спецификации родителя. Внутри закрытой части и теле ребенка видимы приватные объявления родителя:

```
package body Airplane_Package.SST_Package is
   procedure Set_Speed(A: in out SST_Data; I: in Integer) is
    begin
      A.Speed := I;                          -- Правильно, приватное поле в родителе
     end Set_Speed;
end Airplane_Package.SST_Package;
```

Конечно, общая часть ребенка не может обращаться к закрытой части родителя, иначе ребенок мог бы раскрыть секреты родительского пакета.

15.3. Данные класса

Конструкторы и деструкторы

Конструктор (constructor) — это подпрограмма, которая вызывается, когда создается объект класса; когда объект уничтожается, вызывается *деструктор (destructor).* Фактически, каждый объект (переменная), определенный в каком-либо языке, требует выполнения некоторой обработки при создании и уничтожении переменной хотя бы для выделения и освобождения памяти. В объектно-ориентированных языках программист может задать такую обработку.

Конструкторы и деструкторы в языке C++ могут быть определены для любого класса; фактически, если вы не определяете их сами, компилятор обеспечит предусмотренные по умолчанию. Синтаксически конструктор — это подпрограмма с именем класса, а деструктор — то же имя с префиксным символом «~»:

```
class Airplanes {
private:
    Airplane_Data database [100];
    int current_airplanes;
public:
    Airplanes(int i = 0): current_airplanes(i) {};
    ~Airplanes();
};
```

C++

После создания базы данных Airplanes число самолетов получает значение параметра i, который по умолчанию имеет значение ноль:

```
Airplanes a1 (15);                  // current_airplanes =15
Airplanes a2;                       //current_airplanes = O
```

Когда база данных удаляется, будет выполнен код деструктора (не показанный). Можно определить несколько конструкторов, которые перегружаются на сигнатурах параметров:

C++

```
class Airplanes {
public:
    Airplanes(int i = 0): current_airplanes(i) {};
    Airplanes(int i, int j): current_alrplanes(i+j) {};
    ~Airptartes();
};
Airplanes a3(5,6);                  // current_airplanes = 11
```

В языке C++ также есть *конструктор копирования (copy constructor)*, который дает возможность программисту задать свою обработку для случая, когда объект инициализируется значением существующего объекта или, в более общем случае, когда один объект присваивается другому. Полное определение конструкторов и деструкторов в языке C++ довольно сложное; более подробно см. гл. 12 справочного руководства по языку C++.

В языке Ada 95 явные конструкторы и деструкторы обычно не объявляются. Для простой инициализации переменных достаточно использовать значения по умолчанию для полей записи:

```
type Airplanes is tagged
    record
        Current_Airplanes: Integer := 0;
     end record;
```

или дискриминанты (см. раздел 10.4):

Ada

```
type Airplanes(lnitial: Integer) is tagged
    record
       Current_Airplanes: Integer := Initial;
     end record;
```

Программист может определить свои обработчики, порождая тип из абстрактного типа, называемого управляемым (Controlled). Этот тип обеспечивает абстрактные подпрограммы для Инициализации (Initialization), Завершения (Finalization) и Корректировки (Adjust) для присваивания, которые вы можете заместить нужными вам программами. За деталями нужно обратиться к пакету Ada. Finalization, описанному в разделе 7.6 справочного руководства по языку Ada.

Class-wide-объекты

Память распределяется для каждого экземпляра класса:

```
class C {
    chars[100];
};
        C c1,c2;                    //по 100 символов для c1 и c2
```

C++

Иногда полезно иметь переменную, которая является общей для всех экземпляров класса. Например, чтобы присвоить порядковый номер каждому экземпляру, можно было бы завести переменную last для записи последнего присвоенного номера. В языке Ada это явно делается с помощью включения обычного объявления переменной в теле пакета:

```
package body P is
   Last: Integer := 0;
end P;
```

Ada

в то время как в языке'C++ нужно воспользоваться другим синтаксисом:

```
class C {
      static int last;              //Объявление
      chars[100];
};
    int C::last = 0;                // Определение, доступное за пределами файла
```

C++

Спецификатор static в данном случае означает, что будет заведен один CW-объект*. Вы должны явно определить компонент static за пределами определения класса. Обратите внимание, что статический (static) компонент класса имеет внешнее связывание и может быть доступен из других файлов, в отличие от статического объявления в области файла.

Преобразование вверх и вниз

В разделе 14.4 мы описали, как в языке C++ значение порожденного класса может быть неявно преобразовано в значение базового класса. Это называется *преобразованием вверх (up-conversion),* потому что преобразование делается вверх от потомка к любому из его предков. Это также называется *сужением (narrowing),* Потому что производный тип «широкий» (так как он имеет дополнительные поля), в то время как базовый тип «узкий», он имеет только поля, которые являются общими для всех типов в производном семействе. Запомните, что преобразование вверх происходит только, когда значение производного типа непосредственно присваивается переменной базового типа, а не когда указатель присваивается от одной переменной другой.

Преобразование вниз (down-conversion) от значения базового типа к значению производного типа не допускается, поскольку мы не знаем, какие значения включить в дополнительные поля. Рассмотрим, однако, указатель на базовый тип:

```
Base_Class*     Base_Ptr = new Base_Class;
Derived_Class*  Derived_Ptr = new Derived_Class;
```

C++

```
if (...) Base_Ptr = Derived_Ptr;
Derived_Ptr = Base_Ptr;             // На какой тип указывает Base_Ptr?
```

Конечно, возможно, что Base_Ptr фактически укажет на объект производного типа; в этом случае нет никакой причины отклонить присваивание. С другой стороны, если указуемый объект фактически имеет базовый тип, мы делаем попытку преобразования вниз, и присваивание должно быть отвергнуто. Чтобы предусмотреть этот случай, в языке C++ определено *динамическое преобразование типов (dynamic cast)*, которое является условным в зависимости от типа указуемого объекта:

```
Derived_Ptr = dynamic_cast<Derived_Class*>Base_Ptr;
```
C++

Если указуемый объект фактически имеет производный тип, преобразование завершается успешно. В противном случае указателю присваивается 0, и программист может это проверить.

Уже в языке Ada 83 допускалось явное преобразование между любыми двумя типами, порожденными друг из друга. Это не вызывало никаких проблем, потому что производные типы имеют в точности те же самые компоненты. Для них допустимо иметь различные представления (см. раздел 5.8), но преобразование типов совершенно четко определено, потому что оба представления имеют одинаковые число и типы компонентов.

Расширение преобразования производного типа до теговых типов не вызывает проблем в случае преобразования вверх от производного типа к базовому. Ненужные поля усекаются:

```
S:SST_Data;
A: Airplane_Data := Airplane_Data(S);
```
Ada

В другом направлении используются *агрегаты расширения (extention aggregates)*, чтобы обеспечить значения для полей, которые были добавлены при расширении:

```
S:=(AwithMach=>1.7);
```
Ada

Поля Speed и подобные берутся из соответствующих полей в значении A, а дополнительное поле Mach задано явно.

При попытке преобразования вниз CW-типа к конкретному типу делается проверка во время выполнения, и, если CW-объект не производного типа, произойдет исключительная ситуация:

```
I Ada procedure P(C: Airplane_Data'Class) is
      S:SST_Data;
begin
      S := SST_Data(C);                    - Какой тип у C ??
exception
     when Constraint_Error => .. .
end P;
```
Ada

15.4. Язык программирования Eiffel

Основные характеристики языка программирования Eiffel:

• Язык Eiffel изначально создавался как объектно-ориентированный, а не как дополнительная пристройка для поддержки ООП в существующем языке.

• В языке Eiffel программу можно построить единственным способом — как систему классов, которые являются клиентами друга друга или наследуются один из другого.

• Поскольку наследование — это основная конструкция структурирования, центральное место в языке занимает стандартная библиотека классов (связанных наследованием).

• Не будучи частью «языка», развитая среда программирования была создана группой разработчиков языка Eiffel. Среда включает ориентированную на язык поддержку для отображения и изменения классов, для инкрементной компиляции и для тестирования и отладки.

В отличие от языка Smalltalk (который имеет аналогичные характеристики), язык Eiffel жестко придерживается статического контроля соответствия типов наряду с динамическим полиморфизмом, как в языках Ada 95 и C++. Eiffel идет дальше в попытках поддерживать надежное программирование, интегрируя утверждения в язык, как обсуждалось в разделе 11.5.

Единственная программная единица в Eiffel — это класс: никаких файлов, как в языках C и C++, и никаких пакетов, как в языке Ada.

Терминология языка Eiffel отличается от других языков: подпрограммы (процедуры и функции) называются *рутинами (routine),* объекты (переменные и константы) называются *атрибутами (attribute),* а рутины и атрибуты, которые входят в состав класса, называются *свойствами (feature)* класса. По существу, нет никакого различия между функциями и константами: подобно литералу перечисления языка Ada, константа рассматривается просто как функция без параметров. Язык Eiffel статически типизирован, подобно языку C++, в том смысле, что при присваиваниях и при передаче параметров типы должны соответствовать друг другу, и это соответствие может быть проверено во время компиляции. Однако язык не имеет таких богатых конструкций для управления соответствием типов, как подтипы и числовые типы (numerics) языка Ada.

Когда объявляется класс, задается список свойств:

```
class Airplanes
feature                                                    -- "public"
   New_Airplane(Airplane_Data): Integer is
Do
      ….
      end; -- New_Airplane Get_Airplane(lnteger): Airplane_Data is
      do
      ….
      end; -- Get_Airplane
feature {} --"private"
database: ARRAY[Airplane_Data];
current_airpianes: Integer;
find_empty_entry: Integer is
do
    …
end; -- find_empty_entry
end; -- class Airplanes
```

Как и в языке C++, набор свойств может быть сгруппирован, и для каждой такой feature-группы может быть определена своя доступность, feature-группа со спецификатором, который изображает пустое множество «{}», не экспортируется ни в какой другой класс, подобно private-спецификатору, feature-группа без спецификатора экспортируется в любой другой класс в системе; однако это отличается от public-спецификатора в языке C++ и от открытой части спецификации пакета в языке Ada, потому что экспортируется только доступ для чтения. Кроме того, вы можете явно написать список классов в feature-спецификаторе; этим классам будет разрешен доступ к свойствам внутри группы, подобно «друзьям» в языке C++.

В языке Eiffel нет реального различия между предопределенными типами и типами, определенными программистом, database — это объект класса ARRAY, который является предопределенным в библиотеке языка Eiffel. Конечно, «массив» — очень общее понятие; как мы должны указать тип элементов массива? Нужно применить тот же самый метод, который использовал бы программист для параметризации любого типа данных: обобщения

(genetics). Встроенный класс ARRAY имеет один родовой параметр, который используется, чтобы определить тип элементов:

```
class ARRAY[G]
```

Когда объявляется объект типа ARRAY, должен быть задан фактический параметр, в данном случае Airplane_Data. В отличие от языков Ada и C++, которые имеют специальный синтаксис для объявления встроенных составных типов, в языке Eiffel все создается из родовых классов с помощью единого набора синтаксических и семантических правил.

Обобщения широко используются в языке Eiffel, потому что библиотека содержит определения многих родовых классов, которые вы можете специализировать для своих конкретных требований. Родовые классы также могут быть *ограниченными (constrained),* чтобы работала модель контракта между родовым классом и его конкретизацией, как это делается в языке Ada (см. раздел 10.3). Ограничения задаются не сопоставлением с образцом, а указанием имени класса, для которого фактический родовой параметр должен быть производным. Например, следующий родовой класс может быть конкретизирован только типами, производными от REAL:

```
class Trigonometry[R -> REAL]
```

Вы уже заметили, что в классе на языке Eiffel не разделены спецификации свойств и их реализация в виде выполнимых подпрограмм. Все должно находиться в одном и том же объявлении класса, в отличие от языка Ada, который делит пакеты на отдельно компилируемые спецификации и тела. Таким образом, язык Eiffel платит за свою простоту, требуя большего объема работы от среды программирования. В частности, язык определяет *усеченную (short)* форму, по сути интерфейс, и среда отвечает за отображение усеченной формы по запросу.

Наследование

Каждый класс определяет тип, а все классы в системе организованы в одну иерархию. Наверху иерархии находится класс, называющийся ANY. Присваивание и равенство определены внутри ANY, но могут быть замещены внутри класса. Синтаксис для наследования такой же, как в языке C++: унаследованные классы перечисляются после имени класса. Если задан класс Airplane_Data:

```
class Airplane_Data
feature
    Set_Speed(l: Integer) is...
    Get_Speed: Integer is....
feature {}
      ID: STRING;
      Speed: Integer;
     Altitude: Integer;
end;                                    -- class Airplane_Data
```

его можно наследовать следующим образом:

```
class SSTJData inherit
    Airplane_Data
        redefine
            Set_Speed, Get_Speed
        end
```

```
feature
      Set_Speed(l: Integer) is...
      Get_Speed: Integer is...
feature {}
     Mach: Real;
end; — class SST_Data
```

Все свойства в базовом классе наследуются с их экспортируемыми атрибутами в неизменном виде. Однако для производного класса программист может переопределить некоторые или все унаследованные свойства. Переопределяемые свойства должны быть явно перечислены в redefine-конструкции, которая следует за inherit-конструкцией. Кроме переопределения, свойство можно просто переименовать. Обратите внимание, что унаследованное свойство может быть реэкспортировано из класса, даже если оно было приватным в базовом классе (в отличие от языков C++ и Ada 95, которые не разрешают вторгаться в ранее скрытую реализацию).

Среда языка Eiffel может отображать *плоскую (flat)* версию класса, которая показывает все действующие на данный момент свойства, даже если они были унаследованы и повторно объявлены где-то еще в иерархии. Таким образом, интерфейс класса отчетливо отображается, и программисту не нужно «раскапывать» иерархию, чтобы точно увидеть, что было переобъявлено, а что не было.

Eiffel, аналогично языку C++, но, в отличие от языка Ada 95, использует подход отличимого получателя, поэтому нет необходимости задавать явный параметр для объекта, подпрограмма которого должна быть вызвана:

```
A: Airplane_Data;
A.Set_Speed(250);
```

Распределение памяти

В языке EifFel нет никаких явных указателей. Все объекты неявно распределяются динамически и доступны через указатели. Однако программист может по выбору объявить объект как расширенный (expanded), в этом случае он будет размещен и доступен без использования указателя:

```
database: expanded ARRAY[Airplane_Data];
```

Кроме того, класс может быть объявлен как расширенный, и все его объекты будут доступны непосредственно. Само собой разумеется, что встроенные типы Integer, Character и т.д. являются расширенными.

Обратите внимание, что оператор присваивания или проверки равенства

```
X :=Y;
```

дает четыре варианта, в зависимости от того, являются объекты X и Y расширенными оба, либо только один из них, либо ни тот ни другой. В языках Ada и C++ программист отвечает за то, чтобы различать, когда подразумевается присваивание указателя, а когда — присваивание обозначенных объектов. В языке EifFel присваивание прозрачно для программиста, а значение каждого варианта в языке тщательно определено.

Преимущество косвенного распределения состоит в том, что обычные объекты, чей тип есть тип базового класса, могут иметь значения любого типа, чей класс порожден из базового типа:

```
A: Airplane_Data;
S: SST_Data;

A:=S;
```

Если распределение было статическим, в объекте A не будет «места» для дополнительного поля Mach из S. Когда используется косвенное распределение, присваивание — это, по сути, просто копирование указателя. Сравните это с языками Ada 95 и C++, в которых требуются дополнительные понятия: CW-типы и указатели для присваивания, которые поддерживают конкретный тип.
Кроме того, язык Eiffel делает различие между мелким (shallow)и глубоким (deep) копированием в операторах присваивания. При мелком копировании копируются только указатели (или данные, в случае расширенных объектов), в то время как при глубоком копировании копируются структуры данных целиком. Замещая унаследованное определение присваивания, вы можете выбрать любой вариант для любого класса.
Динамический полиморфизм получаем как непосредственное следствие. Возьмем

```
A.Set_Speed(250);
```

Компилятор не имеет никакой возможности узнать, является конкретный тип значения, находящегося в данный момент в A, базовым типом Air-plane_Data для A или некоторым типом, порожденным из Airplane_Data. Так как подпрограмма Set_Speed была переопределена, по крайней мере, в одном порожденном классе, должна выполняться диспетчеризация во время выполнения. Обратите внимание, что не требуется никакого специального синтаксиса или семантики: все вызовы потенциально динамические, хотя компилятор проведет оптимизацию и использует статическое связывание, где это возможно.

Абстрактные классы

Абстрактные классы в языке Eiffel такие же, как в языках C++ и Ada 95. Класс или свойство в классе может быть объявлено как отсроченное (deferred). Отсроченный класс должен быть сделан конкретным при помощи *эффективизации (effecting)* всех отсроченных свойств, т. е. предоставления реализации. Обратите внимание, что, в отличие от языков C++ и Ada 95, вы можете *объявить* объект, чей тип отсрочен; вы получаете null-указатель, который не может использоваться до тех пор, пока ему не будет присвоено значение имеющего силу производного типа:

```
deferred class Set...                  -- Абстрактный класс
class Bit_Set inherit Set...           -- Конкретный класс

S: Set;                                -- Абстрактный объект!
B: Bit_Set;                             - Конкретный объект
!!B;                                   --Создать экземпляр B
S := B;                                -- Правильно,S получает конкретный объект,
S.Union(...);                          --который теперь можно использовать
```

Множественное наследование

Язык Eiffel поддерживает множественное наследование:

```
class Winged_Vehicle
feature
        Weight: Integer;
        display is . .. end;
end;

class Motorized_Vehicle
feature
       Weight: Integer;
       display is ... end;
end;

class Airplane inherit
          Winged_Vehicle, Motorized_Vehicle
          …
end;
```

Поскольку допускается множественное наследование, в языке должно определяться, как разрешить неоднозначности, если имя унаследовано от нескольких предков. Правило языка Eiffel в основе своей очень простое (хотя его формальное определение сложное, поскольку оно должно принимать во внимание все возможности иерархии наследования):

Если свойство унаследовано от класса предка более чем одним путем, оно используется совместно; в противном случае свойства реплицируются.

rename- и redef ine-конструкции могут использоваться для изменения имен по мере необходимости. В примере класс Airplane наследует только одно поле Weight. Очевидно, по замыслу предлагалось для класса иметь два поля Weight, одно для корпуса летательного аппарата и одно для двигателя. Этого можно достичь за счет переименования двух унаследованных объектов:

```
class Airplane inherit
     Winged_Vehicle
          rename Weight as Airframe_Weight;
      Motorized_Vehicle
          rename Weight as Engine_Weight;
      …
end;
```

Предположим теперь, что мы хотим заместить подпрограмму display. Мы не можем использовать redefine, потому что при этом возникла бы неоднозначность указания подпрограммы, которую мы переопределяем. Решение состоит в том, чтобы использовать undefine для отмены определений обеих унаследованных подпрограмм и написать новую:

```
class Airplane inherit
       Winged_Vehicle
            undefine display end;
        Motorized_Vehicle
             undefine display end;
feature
display is... end;
```

end;

В справочном руководстве по языку Eiftel подробно обсуждается использование rename, redefine и undefine для разрешения неоднозначности при множественном наследовании.

15.5. Проектные соображения

Наследование и композиция

Наследование — это только один метод структурирования, который может использоваться в объектно-ориентированном проектировании. Более простым методом является композиция, которая представляет собой вложение одной абстракции внутрь другой. Вы уже знакомы с композицией, поскольку вам известно, что одна запись может быть включена внутрь другой:

```
with Airplane_Package;
package SS"f.Package is
      type SST_Data is private;
private
      type SST_Data is
          record
              A: Airplane. Data;
               Mach: Float;
          end record;
end SST_Package;
```

и в языке C++ класс может включать экземпляр другого класса как элемент:

```
class SST_Data {
private:
   Airplane_Data a;
    float mach;
};
```

Композиция — более простая операция, чем наследование, потому что для ее поддержки не требуется никаких новых конструкций языка; любая поддержка инкапсуляции модуля автоматически дает вам возможности для композиции абстракций. Родовые единицы, которые в любом случае необходимы в языке с проверкой соответствия типов, также могут использоваться для формирования абстракций. Наследование, однако, требует сложной поддержки языка (теговых записей в языке Ada и виртуальных функций в языке C++) и дополнительных затрат при выполнении на динамическую диспетчеризацию.

Если вам нужна динамическая диспетчеризация, то вы должны, конечно, выбрать наследование, а не композицию. Однако, если динамической диспетчеризации нет, выбор зависит только от решения вопроса, какой метод дает «лучший» проект. Вспомните, что язык C++ требует, чтобы при создании базового класса вы решили, должна ли выполняться динамическая диспетчеризация, объявляя одну или несколько подпрограмм как виртуальные; эти и только эти подпрограммы будут участвовать в диспетчеризации. В языке Ada 95 динамическая диспетчеризация потенциально произойдет в любой подпрограмме, объявленной с управляющим параметром тегового типа:

```
type T is tagged ...;
procedure Proc(Parm: T);
```

Фактически решение, является связывание статическим или динамическим, принимается отдельно для каждого вызова. Не используйте наследование, когда подошла бы простая запись.

Основное различие между двумя методами состоит в том, что композиция просто использует существующую закрытую абстракцию, в то время как наследование знает о реализации абстракции. Пользователи закрытой абстракции защищены от изменения реализации. При использовании наследования базовые классы не могут изменяться без учета того, какие изменения это вызовет в производных классах.

С другой стороны, при каждом доступе к закрытой абстракции должна выполняться подпрограмма интерфейса, в то время как наследование разрешает эффективный прямой доступ производным классам. Кроме того, вы можете изменить реализацию в производном классе, в то время как в композиции ограничены использованием существующей реализации. Говоря кратко: легко «купить» и «продать» модули для композиции, в то время как наследование делает вас «партнером» разработчика модуля.

Нет никакой опасности при аккуратном и продуманном использовании любого метода; проблемы могут возникнуть, когда наследование используется беспорядочно, поскольку при этом может возникнуть слишком много зависимостей между компонентами программной системы. Мы оставляем подробное обсуждение относительных достоинств этих двух понятий специализированным работам по ООП. О преимуществах наследования см. книгу Мейера по конструированию объектно-ориентированного программного обеспечения (Meyer, *Object-oriented Software Construction*, Prentice-Hall International, 1988), особенно гл. 14 и 19. Сравните ее с точкой зрения предпочтения композиции, выраженной в статье J.P. Rosen, «What orientation should Ada objects take?» *Communications of the ACM*, 35(11), 1992, стр. 71—76.

Использование наследования

Удобно разделить случаи применения наследования на несколько категорий:

Подобие поведения. SST ведет себя как Airplane. Это простое применение наследования для совместного использования кода: операции, подходящие для Airplane, подходят для SST. Операции при необходимости могут быть замещены.

Полиморфная совместимость. Linked-Set (связанное множество) и Bit-Set (битовое множество) полиморфно совместимы с Set. Происходя от общего предка, множества, которые реализованы по-разному, могут быть обработаны с помощью одних и тех же операций. Кроме того, вы можете создавать разнородные структуры данных, отталкиваясь от предка, который содержит элементы всего семейства типов.

Родовая совместимость. Общие свойства наследуются несколькими классами. Эта методика применяется в больших библиотеках, таких как в языках Smalltalk или Eiffel, где общие свойства выносятся в классы-предки, иногда *называемые аспект-классами (aspect classes).* Например, класс Comparable (сравнимый) мог бы использоваться для объявления таких операций отношения, как «<», и любой такой класс, как Integer или Float, обладающий такими операциями, наследуется из Comparable.

Подобие реализации. Класс может быть создан путем наследования логических функций из одного класса и их реализации — из другого. Классический пример — Bounded_Stack, который (множественно) наследует функциональные возможности из Stack и их реализации из Array. В более общем смысле, класс, созданный множественным наследованием, насле-

довал бы функциональные возможности из нескольких аспект-классов и реализацию из одного дополнительного класса.

Эти категории не являются ни взаимоисключающими, ни исчерпывающими; они представлены как руководство к использованию этой мощной конструкции в ваших программных проектах.

Перегрузка и полиморфизм

Хотя перегрузка (overloading) — это форма полиморфизма («многофор-менности»), эти две концепции применяются в совершенно разных целях. Перегрузка используется как удобное средство для задания одного и того же имени подпрограммам, которые функционируют на различных типах, в то время как динамический полиморфизм используется для реализации операции для семейства связанных типов. Например:

```
void proc put(int);
void proc put(float);
```
C++

представляет перегрузку, потому что общее имя используется только для удобства, и между int и float нет никакой связи. С другой стороны:

```
virtual void set_speed(int):
```
C++

является одной подпрограммой, которая может быть реализована по-разному для разных типов самолетов.
Технически трудно совместить перегрузку и динамический полиморфизм и не рекомендуется использовать эти два понятия вместе. Не пытайтесь внутри порожденного класса перегружать подпрограмму, которая появляется в базовом классе:

```
class SST_Data : public Airplane_Data {
public:
     void set_speed(float);                    //float, а не int
};
```
C++

Правила языка C++ определяют, что эта подпрограмма и не перегружает, и не замещает подпрограмму в базовом классе; вместо этого она *скрывает* определение в базовом классе точно так же, как внутренняя область действия!

Язык Ada 95 допускает сосуществование перегрузки и замещения :

```
with Airplane_Package; use Airplane_Package;
package SST_Package is
      type SSTJData is new Airplane_Data with ...
   procedure Set_Speed(A: in out SST_Data; I: in Integer);
                        -- Замещает примитивную подпрограмму из Airplane_Package
procedure Set_Speed(A: in out SST_Data; I: in Float);
                        -- Перегрузка, не подпрограмма-примитив
end SST_Package;
```
Ada

Поскольку нет примитивной подпрограммы Set_Speed с параметром Float для родительского типа, второе объявление — это просто самостоятельная подпрограмма, которая перегружает то же самое имя. Хотя это допустимо, этого следует избегать, потому что пользователь типа, скорее всего, запутается. Посмотрев только на SST_Package (и без комментариев!), вы не сможете сказать, какая именно подпрограмма замещается, а какая перегружается:

```
procedure Proc(A: Airplane_Data'Class) is
begin
        Set_Speed(A, 500);          -- Правильно, диспетчеризуется
        Set_Speed(A, 500.0);        -- Ошибка, не может диспетчеризоваться!
end Proc;
```
Ada

15.6. Методы динамического полиморфизма

Мы заключаем эту главу подведением итогов по динамическому полиморфизму в языках для объектно-ориентированного программирования.

Smalltalk. Каждый вызов подпрограммы требует динамической диспетчеризации, которая включает поиск по иерархии наследования, пока подпрограмма не будет найдена.

Eiftel. Каждый вызов подпрограммы диспетчеризуется динамически (если оптимизация не привела к статическому связыванию). В отличие от языка Smalltalk, возможные замещения известны во время компиляции, поэтому диспетчеризация имеет фиксированные издержки, вносимые таблицей переходов.

C++. Подпрограммы, которые явно объявлены виртуальными и вызываются косвенно через указатель или ссылку, диспетчеризуются динамически. Диспетчеризация во время выполнения имеет фиксированные издержки.

Ada 95. Динамическая диспетчеризация неявно используется для примитивных подпрограмм тегового типа, когда фактический параметр является CW-типом, а формальный параметр имеет конкретный тип. Затраты на диспетчеризацию во время выполнения фиксированы.

Языки отличаются деталями программирования и затратами, требующимися для динамического полиморфизма, и это влияет на стиль программирования и эффективность программ. Ясное понимание заложенных в языках принципов поможет вам сравнивать объектно-ориентированные языки и разрабатывать и создавать хорошие объектно-ориентированные программы на любом языке, который вы выберете.

15.7. Упражнения

1. Реализуйте пакеты на языке Ada 95 и классы на языке C++ для работы с множествами.

2. Может ли абстрактный тип в языке Ada 95 или абстрактный класс в языке C++ иметь компоненты-данные? Если так, для чего они могли бы использоваться?

```
type Item is abstract tagged
   record
      I: Integer;
    end record;
```
Ada

3. Напишите программу для неоднородной очереди, основываясь на абстрактном классе.

4. Реализуйте пакеты/классы для множеств с родовым типом элемента, а не только для целочисленных элементов.

5. Подробно изучите множественное наследование в языке Eiffel и сравните его с множественным наследованием в языке C++.

6. Стандартный пример множественного наследования в языке Eiffel -список фиксированного размера, реализованный с помощью наследования, как от списка, так и от массива. Как бы вы написали такие ADT (абстрактные типы данных) на языке Ada 95, в котором нет множественного наследования?

7. Чем опасно определение защищенных (protected) данных в языке C++? Относится ли это также к пакетам-детям в языке Ada 95?

7. Изучите структуру стандартной библиотеки в языке Ada 95, в котором широко используются пакеты-дети. Сравните ее со структурой стандартных классов ввода-вывода в языке C++.

9. Изучите пакет Finalization в языке Ada 95, который может использоваться для написания конструкторов и деструкторов. Сравните его с конструкциями языка C++.

10. Какова связь между операторами присваивания и конструкторами/де структорами?

11. Дайте примеры использования CW-объектов.

5 Непроцедурные языки программирования

Глава 16
Функциональное программирование

16.1. Почему именно функциональное программирование?

В разделе 1.8 мы упоминали, что и Черч и Тьюринг предложили модели для вычислений задолго до того, как появились первые компьютеры. Машины Тьюринга очень похожи на современные компьютеры тем, что они основаны на *обновляемой* памяти, т. е. наборе ячеек памяти, содержимое которых изменяется при выполнении команд. Это также известно как архитектура *фон Неймана*.

Формулировка модели вычислений Черча (названная *лямбда-исчислением)* совершенно другая — она основана на математическом понятии функции. Эта формулировка полностью эквивалентна формулировке Тьюринга в смысле представления вычислений, которые могут быть точно описаны, но в качестве формализма, применяемого для вычислений на практике, функциональный подход всегда был менее популярен. В языке Lisp, разработанном в 1956 г., для вычислений используется функциональный подход, подобный модели лямбда-исчисления, хотя многие его особенности поощряют стиль процедурного программирования.

В 1980-е годы дальнейшие исследования в области функционального программирования привели к разработке языков на чисто теоретических основаниях, которые тем не менее могут быть эффективно реализованы. Основное различие между современными функциональными языками программирования и языком Lisp состоит в том, что в них типы и контроль соответствия типов являются базисными понятиями, поэтому значительно возросли и надежность, и эффективность программ.

Многие проблемы, с которыми мы сталкиваемся при написании надежной программы, возникают непосредственно из-за использования обновляемой памяти:

• Память может быть «затерта», потому что мы непосредственно изменяем ячейки памяти (используя индексы массива или указатели), а не просто вычисляем значения.

• Трудно создавать сложные программы из компонентов, потому что подпрограммы могут иметь побочные эффекты. Поэтому может оказаться, что даже осознать все последствия работы подпрограммы нельзя в отрыве от всей остальной программы.

Строгий контроль соответствия типов и методы инкапсуляции объектно-ориентированного программирования могут смягчить эти проблемы, но не могут устранить их полностью. При функциональном подходе обе эти проблемы исчезают.

Дальнейшее обсуждение будет базироваться на популярном языке Standart ML, хотя эти понятия справедливы и для других языков.

16.2. Функции

Функции определяются в языке ML заданием равенства между именем функции с формальным параметром и выражением:

```
fun even n = (n mod 2 = 0)
```

Различие состоит в том, что здесь нет никаких глобальных переменных, никакого присваивания, никаких указателей и, следовательно, никаких побочных эффектов. После того как функция была определена, она может быть *применена (applied),* и *вычисление (evaluation)* ее применения и дает результат:

```
even 4 = true
even 5 = false
```

С каждой функцией связан тип, точно так же, как в языках программирования типы связаны с переменными. Тип функции even (четное) задается следующим образом:

```
even: int -> bool
```

Это означает, что она отображает значение целочисленного типа в значение булева типа. В языке ML выражения могут содержать условия:

```
fun min (x,y) = if x < y then x else y
```

Приведем пример вычисления при применении функции:

```
min (4,5) =
(if x < y then x else y) (4,5) =
if 4 < 5 then 4 else 5 =
if true then 4 else 5 =
4
```

Обратите внимание, что это не if-оператор, а условное выражение, аналогичное имеющемуся в языке C:

```
x< y?x:y
```

Какой тип у min? В функциональном программировании считается, что функция имеет в точности один аргумент, если же вам требуется большее число аргументов, вы должны создать кортеж (двойной, тройной и т.д.), используя функцию декартова произведения. Таким образом, (4,5) имеет тип int x int, а функция min имеет тип:

```
min: (int x int) -> int
```

Вместо кортежей вы можете определить функцию, которая будет применяться к каждому аргументу по очереди:

```
fun min_c x y = if x < y then x else y
```

Это *карризованная функция (curriedfunction*, от имени математика Н.В. Сипу). Когда эта функция применяется к последовательности аргументов, при первом обращении создается другая функция, которая затем применяется ко второму аргументу.
Функция min_c берет один целочисленный аргумент и создает новую функцию, также с одним аргументом:

```
min_c 4 = if 4 < y then 4 else y
```

Эта функция может затем применяться к другому одиночному аргументу:

```
min_c 4 5 =
(if 4 < y then 4 else y) 5 =
if 4 < 5 then 4 else 5 =
if true then 4 else 5 =
4
```

Карризованные функции могут использоваться в *частичных вычислениях* для определения новых функций:

```
fun min_4 = min_c 4
```

```
min_4 5 =
(if 4 < y then 4 else y) 5 =
if 4 < 5 then 4 else 5 =
if true then 4 else 5 =
4
```

16.3. Составные типы

Списки

Список можно создать из элементов любого предварительно определенного типа, в частности из встроенных типов, например целого или булева. Следующие списки:

```
[2,3,5,7,11 ]                    [true, false, false]
```

имеют типы int list и bool list, соответственно. Список можно создать также с помощью *конструкторов (constructors);* конструкторы списка — это [] для пустого списка, и element::list для непустого списка, создаваемого добавлением элемента (element) к существующему списку (list). Конструкторы могут использоваться при определении функций путем сопоставления с образцом:

```
fun          member [] e = false
|            member [e :: tail] e = true
j            member [e1 :: tail] e = member tail e
```

Тип функции member (член) определяется как:

```
member: int list x jnt -> boolean
```

и это можно прочитать следующим образом:

Когда функция member применяется к списку L, а (затем) к элементу e, вычисление основывается на вариантах выбора в зависимости от аргументов: 1) если L пуст, e не является членом L; 2) если e — первый элемент L, то e является членом L; 3) в противном случае, e1, первый элемент списка L, отличен от e, и мы (рекурсивно) проверяем, является ли e членом оставшейся части списка L.

В языке ML вам не нужно объявлять тип функции; компилятор автоматически выводит тип функции из типов аргументов и типа результата. Если компилятор не может вывести тип, вам придется задать нужное количество объявлений типа, чтобы ликвидировать неопределенность выражения. Контроль соответствия типов статический, поэтому, когда функция применяется к значению, проверка соответствия типа функции типу параметра делается во время компиляции.

Обратите внимание, что эта функция рекурсивная. Рекурсия чрезвычайно важна в функциональных языках программирования; при отсутствии «операторов» это единственный способ создания циклов при вычислении выражения.

В качестве заключительного примера покажем, как на языке ML написать алгоритм *сортировки вставкой (insertion sort).* Вы используете этот алгоритм для сортировки при игре в карты: по очереди берете карты из колоды и кладете их в нужное место:

```
fun     insertion_sort [] = []
|       insertion_sort head:: tail =
        insert_element head insertion_sort tail
and
fun     insert_element x [] = [x]
|       insert_element x head:: tail =
        if x < head then x::head:: tail
        else head:: (insert_element x tail)
```

Эти функции имеют типы:

```
insertion_sort: int list -> int list
insert_element: int -> int list -> int list
```

Как только вы привыкнете к этой записи, читать такие программы будет просто:

Отсортированный пустой список является пустым. При сортировке непустого списка берется первый элемент x, сортируется оставшаяся часть списка tail, а затем x вставляется в подходящее место отсортированного списка.

Элемент x вставляется в пустой список, создавая список из одного элемента. Чтобы вставить x в непустой список, нужно сравнить x с первым элементом (head) списка: 1) если x меньше head, сделать x новым первым элементом списка; 2) в противном случае создать новый список, составленный из head, за которым следует остаток списка, в который вставлен x.

Обратите внимание, что -> имеет правую ассоциативность:

```
insert_element: int -> (int list -> int list)
```

Функция отображает целое число в другую функцию, которая отображает список целых в список целых. При помощи частичных вычислений мы можем создавать новые функции подобно следующей:

```
fun insert_4 = insert_element 4
```

которая вставляет 4 в список целых.

В отличие от процедурной программы для того же самого алгоритма здесь нет никаких индексов и никаких for-циклов. Кроме того, ее можно легко обобщить для сортировки объектов других типов, просто заменяя операцию «<» соответствующей булевой функцией для сравнения двух значений рассматриваемого типа. Чтобы создать список, явные указатели не нужны; они заданы неявно в представлении данных. Конечно, сортировка списков в любом языке не так эффективна, как сортировка массива «на своем месте», но для многих приложений, использующих списки, вполне практична.

Определение новых типов

Повсюду в этой книге мы видели, что определение новых типов весьма существенно, если язык программирования должен моделировать реальные объекты. Современные функциональные языки программирования также имеют эту возможность. Определим (рекурсивный) тип для деревьев, узлы которых помечены целыми числами:

```
datatype int tree =
        Empty
|       T of (int tree x int x int tree)
```

Это следует читать так:

int tree является новым типом данных, значения которого: 1) новое константное значение Empty (пусто) или 2) значение, которое сформировано *конструктором* T, примененным к тройке, состоящей из дерева, целого числа и другого дерева.

Определив новый тип, мы можем написать функции, которые обрабатывают дерево.

Например:

```
fun     sumtree Empty = O
|       sumtree T(left, value, right) =
        (sumtree left) + value + (sumtree right)
```

суммирует значения, помечающие узлы дерева, и:

```
fun     mintree Empty = maxint
|       mintree T(left, value, right) =
            min left (min value (mintree right))
```

вычисляет минимальное из всех значений, помечающих узлы, возвращая наибольшее'целое число maxint для пустого дерева.

Все стандартные алгоритмы обработки деревьев можно написать аналогичным образом: определить новый тип данных, который соответствует древовидной структуре, а затем написать функции для этого типа. При этом никакие явные указатели или циклы не нужны, «работают» только рекурсия и сопоставление с образцом.

16.4. Функции более высокого порядка

В функциональном программировании функция является обычным объектом, имеющим тип, поэтому она может быть аргументом других функций. Например, мы можем создать обобщенную (родовую) форму для insert_element (вставить элемент), просто добавляя функцию compare как дополнительный аргумент:

```
fun       general_insert_element compare x [ ] = [x]
|          general_insert_element compare x head:: tail =
            if compare x head
            then x::head::tail
            else head:: (general_insert_element compare x tail)
```

Если string_compare является функцией от string к boolean:

string_compare: (string x string)—> bool

применение general_insert_element к этому аргументу:

fun string_insert = general_insert_element string_compare

дает функцию следующего типа:

string -> string list -> string list

Обратите внимание, что, в отличие от процедурных языков, это обобщение достигается естественно, без какого-либо дополнительного синтаксиса или семантики, наподобие generic или template.

Но какой тип у general_insert_element? Первый аргумент должен иметь тип «функция от пары чего-либо к булеву значению», второй аргумент должен иметь тип этого самого «чего-либо», а третий параметр является списком этих «чего-либо». *Типовые переменные (type variables)* используются в качестве краткой записи для этого «чего-либо», и, таким образом, тип функции будет следующим:

general_insert_element: (('t x 't) -> bool) -> 't -> 't list

где типовые переменные записаны в языке ML как идентификаторы с предшествующим апострофом

Использование *функций более высокого порядка*, т. е. функций, аргументами которых являются функции, не ограничено такими статическими конструкциями, как обобщения. Чрезвычайно полезная функция — map:

```
fun       map f [] = [ ]
|          map f head :: tail = (f head):: (map f tail)
```

Эта функция применяет первый аргумент к списку значений, производя список результатов. Например:

map even [1, 3, 5, 2, 4, 6] = [false, false, false, true, true, true]
map min [(1,5), (4,2), (8,1)] = [1,2,1]

Этого фактически невозможно достичь в процедурных языках; самое большее, мы могли бы написать подпрограмму, которая получает указатель на функцию в качестве аргумента, но мы потребовали бы разных подпрограмм для каждой допустимой сигнатуры аргумента функции.

Обратите внимание, что эта конструкция надежная. Тип map следующий:

```
map: (t1 -> t2) -> 't1 list -> t2 list
```

Это означает, что элементы списка аргументов t1 list все должны быть совместимы с аргументом функции t1, а список результата t2 list будет состоять только из элементов, имеющих тип результата функции t2.

Функции более высокого порядка абстрагируются от большинства управляющих структур, которые необходимы в процедурных языках. В другом примере функция accumulate реализует «составное» применение функции, а не создает список результатов, подобно map:

```
fun       accumulate f initial [] = initial
|         accumulate f initial head::tail - accumulate f (f initial head) tail
```

Функция accumulate может использоваться для создания ряда полезных функций. Функции

```
fun minlist = accumulate min maxint
fun sumlist = accumulate "+" 0
```

вычисляют минимальное значение целочисленного списка и сумму целочисленного списка соответственно. Например:

```
minlist [3, 1,2] =
accumulate min maxint [3, 1,2] =
accumulate min (min maxint 3) [1,2] =
accumulate min 3 [1, 2] =
accumulate min (min 3 1) [2] =
accumulate min 1 [2] =
accumulate min (min 1 2) [] =
accumulate min 1 [] =
1
```

Функции более высокого порядка не ограничиваются списками; можно написать функции, которые обходят деревья и применяют функцию в каждом узле. Кроме того, функции могут быть определены на *типовых* переменных так, чтобы их можно было использовать без изменений при определении новых типов данных.

16.5. Ленивые и жадные вычисления

В процедурных языках мы всегда предполагаем, что фактические параметры вычисляются до вызова функции:

C

```
n = min (j + k, (i + 4) /m);
```

Для обозначения такой методики используется термин — *жадные вычисления*. Однако жадные вычисления имеют свои собственные проблемы, с которыми мы столкнулись в if-операторах (см. раздел 6.2), когда пришлось определить специальную конструкцию для укороченного вычисления:

Ada

```
if (N > 0) and then ((Sum / N) > M) then . . .
```

Как должно быть определено условное выражение

```
if c then e1 else e2
```

в функциональном языке программирования? При жадном вычислении мы вычислили бы c, e1 и e2 и только затем выполнили условную операцию. Конечно, это неприемлемо: следующее выражение нельзя успешно выполнить, если используются жадные вычисления, так как невозможно взять первый элемент пустого списка:

```
if list = [] then [] else hd list
```

Чтобы решить эту проблему, в язык ML введено специальное правило для вычисления if-функции: сначала вычисляется условие c, и только после этого вычисляется один из двух вариантов.

Ситуация была бы намного проще, если бы использовались *ленивые вычисления*, где аргумент вычисляется только, когда он необходим, и только в нужном объеме.

Например, мы могли бы определить if как обычную функцию:

```
fun     if true x y = x
|       if false x y = y
```

Когда применяется if, функция просто применяется к первому аргументу, производя следующее:

```
(if list = [] [] hd list) [] =
if [] = [] [] hd [] =
if true [] hd [] =
[]
```

и мы не пытаемся вычислить hd [].

Ленивое вычисление аналогично механизму вызова параметра по имени (call-by-name) в процедурных языках, где фактический параметр вычисляется каждый раз заново, когда используется формальный параметр. Этот механизм в процедурных языках сомнителен из-за возможности побочных эффектов, которые не позволяют сделать оптимизацию путем вычисления и сохранения результата для многократного использования. В функциональном программировании, свободном от побочных эффектов, такой проблемы нет, и языки, использующие ленивые вычисления (например, Miranda), были реализованы. Ленивые вычисления могут быть менее эффективными, чем жадные, но у них есть значительные преимущества.

В ленивых вычислениях больше всего привлекает то, что можно делать пошаговые вычисления и использовать их, чтобы запрограммировать эффективные алгоритмы. Например, рассмотрим дерево целочисленных значений, чей тип мы определили выше. Вы можете запрограммировать алгоритм, который сравнивает два дерева, чтобы выяснить, имеют ли они одинаковый набор значений при некотором упорядочении узлов. Это можно записать следующим образом:

```
fun equal_nodes t1 t2 = compare_lists (tree_to_list t1) (tree_to_list t2)
```

Функция tree_to_list обходит дерево и создает список значений в узлах; com-pare_lists проверяет, равны ли два списка. При жадных вычислениях оба дерева полностью преобразуются в списки до того, как будет выполнено сравнение, даже если при обходе выяснится, что первые узлы не равны! При ленивых вычислениях функции нужно вычислять только в том объеме, который необходим для ответа на поставленный вопрос.

Функции compare_lists и tree_to_list определены следующим образом:

```
fun          compare_lists [] [] = true
|            compare_lists head::tail1 head::tail2 = compare_lists tail1 tail2
|            compare_lists list1 Iist2 = false
fun          tree_to_list Empty = []
|            tree_to_listT(left, value, right) =
                    value :: append (tree_to_list left) (tree_to_list right)
```

Ленивые вычисления, например, могли бы происходить следующим образом (мы использовали сокращенные имена функций cmp и ttl, а многоточием обозначили очень большое поддерево):

```
cmp      ttl T(T(Empty,4,Empty), 5, . . .)
         ttl T(T(Empty,6,Empty), 5,...) =
cmp       5:: append (ttl T(Empty,4,Empty)) (ttl.. .)
          5:: append (ttl T(Empty,6,Empty)) (ttl.. .) =
cmp      append (ttl T(Empty,4,Empty)) (ttl.. .)
          append (ttl T(Empty,6,Empty)) (ttl. ..) =
           …
cmp       4:: append [] (ttl. . .)
          6:: append [] (ttl. ..) =
          false
```

Вычисления, выполняемые только по мере необходимости, позволили полностью избежать ненужного обхода правого поддерева. Чтобы достичь того же самого эффекта в языке, подобном ML, который использует жадные вычисления, придется применять специальные приемы.

Дополнительное преимущество ленивых вычислений состоит в том, что они подходят для интерактивного и системного программирования. Ввод, скажем, с терминала рассматривается просто как потенциально бесконечный список значений. При ленивых вычислениях, конечно, никогда не рассматривается весь список: вместо этого всякий раз, когда пользователь вводит значение, забирается первый элемент списка.

16.6. Исключения

Вычисление выражения в языке ML может привести к исключению. Существуют стандартные исключения, которые в основном возникают при вычислениях со встроенными типами, например при делении на ноль или попытке взять первый элемент пустого списка. Программист также может объявлять исключения, у которых могут быть необязательные параметры:

```
exception BadParameter of int;
```

После этого может возникнуть исключение, которое можно обработать:

```
fun only_positive n =
          if n <= 0 then raise BadParameter n
          else...
val i = ...;
val j = only_positive i
      handle
        BadParameter 0 => 1;
        BadParameter n => abs n;
```

Функция only_positive возбудит исключение BadParameter, если параметр не положителен. При вызове функции обработчик исключения присоединяется к вызывающему выражению, определяя значение, которое будет возвращено, если исключение наступит. Это значение можно использовать для дальнейших вычислений в точке, где произошло исключение; в этом случае оно используется только как значение, возвращаемое функцией.

16.7. Среда

Помимо определений функций и вычисления выражений программа на языке ML может содержать объявления:

```
val i = 20
val'S = "Hello world"
```

Таким образом, в языке ML есть память, но, в отличие от процедурных языков, эта память необновляемая; в данном примере нельзя «присвоить» другое значение i или s. Если мы теперь выполним:

```
val i = 35
```

будет создано новое именованное значение, скрывающее старое значение, но не заменяющее содержимое i новым значением. Объявления в языке ML аналогичны объявлениям const в языке C в том смысле, что создается объект, который нельзя изменить; однако повторное определение в языке ML скрывает предыдущее, в то время как в языке C запрещено повторно объявлять объект в той же самой области действия.

Блочное структурирование можно делать, объявляя определения или выражения как локальные. Синтаксис для локализации внутри выражения показан в следующем примере, который вычисляет корни квадратного уравнения, используя локальное объявление для дискриминанта:

```
val a = 1.0 and b = 2.0 and c = 1.0
let
   D = b*b-4.0*a*c
in
   ((- b + D)/2.0*a, (- b - D)/2.0*a )
end
```

Каждое объявление *связываем (binds)* значение с именем. Набор всех таких связей, действующих в какой-либо момент времени, называется *средой (environment),* и мы говорим, что выражение вычислено в контексте среды. Мы фактически обсуждали среды детально в контексте областей действия и видимости в процедурных языках; различие состоит в том, что связывания не могут изменяться в среде функционального программирования.

Очень просто выглядит расширение, позволяющее включать в среду абстрактные типы данных. Это делается присоединением набора функций к объявлению типа:

```
abstype int tree =
        Empty
|       T of (int tree x int x int tree)
with
           fun sumtree t = . . .
           fun equal_nodes t1 t2 = .. .
end
```

Смысл этого объявления состоит в том, что только перечисленные функции имеют доступ к конструкторам абстрактного типа аналогично приватному типу в языке Ada или классу в языке C++ с приватными (private) компонентами. Более того, абстрактный тип может быть параметризован *типовой* переменной:

```
abstype 't tree = . . .
```

что аналогично созданию родовых абстрактных типов данных в языке Ada.

Язык ML включает очень гибкую систему для определения и управления модулями. Основное понятие — это *структура (structure),* которая инкапсулирует среду, состоящую из объявлений (типов и функций), аналогично классу в языке C++ или пакету в языке Ada, определяющему абстрактный тип данных. Однако в языке ML структура сама является объектом, который имеет тип, названный *сигнатурой (signature).* Структурами можно управлять, используя *функторы (functors),* которые являются функциями, отображающими одну структуру в другую. Это — обобщение родового понятия, которое отображает пакет или шаблоны класса в конкретные типы. Функторы можно использовать, чтобы скрыть или совместно использовать информацию в структурах. Детали этих понятий выходят за рамки данной книги, и заинтересованного читателя мы отсылаем к руководствам по языку ML.

Функциональные языки программирования можно использовать для написания кратких, надежных программ для приложений, имеющих дело со сложными структурами данных и сложными алгоритмами. Даже если эффективность функциональной программы неприемлема, ее все же можно использовать как прототип или рабочую спецификацию для создаваемой программы.

16.8. Упражнения

1. Какой тип у карризованной функции min_c?

```
fun min_c x y = if x < y then x else y
```

2. Выведите типы sumtree и mintree.

3. Опишите словами определение general_insert_element.

4. Напишите функцию для конкатенации списков, а затем покажите, как ту же самую функцию можно определить с помощью accumulate.

5. Напишите функцию, которая берет список деревьев и возвращает список минимальных значений в каждом дереве.

6. Выведите типы compare_lists и tree_to_list.

7. Что делает следующая программа? Какой тип у функции?

```
fun                    filter f[] = []
|     filter f h ::t = h:: (filter f t),          if f h = true
|     filter f h :: t = filter f t,               otherwise
```

8. Стандартное отклонение последовательности чисел *(x\, . .. , x„)* определяется как квадратный корень из среднего значения квадратов чисел минус квадрат среднего значения. Напишите программу на языке ML, которая вычисляет стандартное отклонение списка чисел. Подсказка: используйте tap и accumulate.

9. Напишите программу на языке ML для поиска совершенных чисел; $n > 2$ — совершенное число, если множители я (не включая *n)* в сумме дают *n*. Например, 1 +2 + 4 + 7+ 14 = 28.В общих чертах программа будет выглядеть как:

```
fun isperfect n =
    let fun addfactors...
in addfactors(n div 2) = n end;
```

10. Сравните исключения в языке ML с исключениями в языках Ada, C++ и Eiffel.

Глава 17

Логическое программирование

Логическое программирование основано на том наблюдении, что формулы математической логики можно интерпретировать как спецификацию вычисления. Стиль программирования при этом становится скорее декларативным, чем процедурным. Мы не выдаем команды, сообщающие компьютеру, что делать; вместо этого мы описываем связь между входными и выходными данными и предоставляем компьютеру «догадаться», как получить из входа выход. В пределах, в которых этого удается достичь, логическое программирование обеспечивает значительно более высокий уровень абстракции с соответствующим преимуществом чрезвычайной краткости программ.

Есть две основные абстракции, которые характеризуют логическое программирование. Суть первой состоит в том, что от таких управляющих операторов, как хорошо известные for и if, мы отказываемся полностью. Вместо них «компилятор» предоставляет чрезвычайно мощный механизм управления, который *единообразно* применяется ко всей программе. Механизм основан на понятии *доказательства* в математической логике: программа рассматривается не как пошаговый алгоритм, а как набор логических формул, которые предполагаются истинными (аксиомы), а вычисление — как попытка доказать формулу на основе аксиом программы.

Суть второй абстракции в том, что больше не используются операторы присваивания и явные указатели; вместо этого для создания и декомпозиции структур данных используется обобщенный механизм сопоставления с образцом, названный *унификацией*. При унификации создаются неявные указатели на компоненты структур данных, но программист видит только абстрактные структуры данных, такие как списки, записи и деревья.

После того как мы обсудим «чистое» логическое программирование, мы опишем компромиссы, введенные в языке Prolog, первом и все еще очень популярном языке логического программирования, используемом на практике.

В то время как проделать вручную вышеупомянутое вычисление довольно утомительно, для компьютера это всего лишь случай без конца повторяющейся задачи, с которой он превосходно справляется. Для выполнения логической программы набор логических формул (программа) и формула-цель, например:

"wor" =>"Hello world"?

предлагаются программной системе, которая названа *машиной вывода*, потому что она из одних формул выводит другие, являющиеся их следствием, пока проблема не будет решена. Метод логического вывода проверяет, можно ли доказать целевую формулу, исходя из аксиом, т.е. формул программы, которые приняты за истинные. Ответом может быть и «да», и «нет», что в логическом программировании называется «успехом» или «неуспехом». «Неуспех» мог быть получен из-за того, что цель не следует из программы, например, "wro" не является подстрокой "Hello world", или из-за неправильности программы, например, если мы пропустили одну из формул программы. Возможен и третий вариант, когда поиск машины вывода будет продолжаться без выбора того или иного ответа, и, так же как это может случиться с while-циклом в языке C, никогда не завершится.

Основные понятия логического программирования:

• Программа является декларативной и состоит исключительно из формул математической логики.

• Каждый набор формул для того же самого *предиката* (такого как «с») интерпретируется как процедура (возможно, рекурсивная).

• Конкретное вычисление определяется предложенной *целью*, т.е. формулой, о которой нужно выяснить, является ли она следствием программы.

• Компилятор является машиной вывода, которая по мере возможности *ищет* доказательство цели из программы.

Таким образом, каждую логическую программу можно прочитать двояко: как набор формул и как спецификацию вычисления. В некотором смысле, логическая программа — это минимальная программа. В разработке программного обеспечения вас обучают точно определять смысл программы перед попыткой ее реализовать, и для точной спецификации используется формальная нотация, обычно некоторая форма математической логики. Если спецификация является программой, то делать больше нечего, и тысячи программистов можно заменить горсткой логиков. Причина того, что логическое программирование нетривиально, состоит в том, что чистая логика недостаточно эффективна для практического программирования, и поэтому есть этап, который должен быть пройден от научной теоретической логики до ее инженерных приложений в программировании.

В логическом программировании нет никаких «операторов присваивания», потому что управляющая структура единообразна для всех программ и состоит из поиска доказательства формулы. Поиск решений проблемы, конечно, не нов; новым является предположение, что поиск решений вычислительных проблем возможен в рамках общей схемы логических доказательств. Логика стала логическим программированием, когда было обнаружено, что, ограничивая структуру формул и способы, которыми делается поиск доказательств, можно сохранить простоту логических утверждений и тем не менее искать решения проблем эффективным способом. Перед объяснением как это делается, мы должны обсудить, как обрабатываются данные в логическом программировании.

17.2. Унификация

Хорновский клоз (Hom clause) — это формула, в которой с помощью *конъюнкции* («и»)элементарных формул выводится одиночная элементарная формула:

(s=>t)<=(t = tl||t2)^(S=>tl)

Логическое программирование основано на том наблюдении, что, ограничивая формулы хорновскими клозами, мы получаем правильное соотношение между выразительностью и эффективностью вывода. Такие факты, как t => t, являются выводами, которые ниоткуда не следуют, т. е. они всегда истинны. Вывод также называется *головой* формулы, потому при записи в инверсной форме оно появляется в формуле первым.
Чтобы инициализировать вычисление логической программы, задается *цель:*

"WOF" => "Hello world"?

Машина вывода пытается сопоставить цель и вывод формулы. В данном случае соответствие устанавливается сразу же: "wor" соответствует переменной s, а "Hello world" — переменной t. Это определяет *подстановку* выражений (в данном случае констант) для переменных; подстановка применяется ко всем переменным в формуле:
"wor" с "Hello world" с= ("Hello world" = tl || t2) л ("wor" с tl)

Теперь мы должны показать, что:

("Hello world" = t1|| t2) л ("wor" с tl)

является истинным, и это ведет к новому соответствию образцов, а именно попытке установить соответствие "Hello world" с tl || t2. Здесь, конечно, может быть много соответствий, что приведет к поиску. Например, машина вывода может допускать, чтобы tl указывало на "Не", а t2 указывало на "Но world"; эти подстановки затем проводятся во всем вычислении.

Знак «: — » обозначает импликацию, а переменные должны начинаться с прописных букв. Когда задана цель:

?- substring ("wor", "Hello world").

вычисление пытается унифицировать ее с головой формулы; если это удается сделать, цель заменяется последовательностью элементарных формул (также называемых целями):

?- concat ("Hello world", T1,*12),* substring ("wor", T1).

Цель, которая получается в результате, может состоять из боле? чем одной элементарной формулы; машина вывода должна теперь выбрать одну из них, чтобы продолжить поиск решения. По *правилу вычисления* языка Prolog машина вывода всегда выбирает *крайнюю левую* элементарную формулу. В данном примере правило вычисления требует, чтобы concat было выбрано перед рекурсивным вызовом substring.

Головы нескольких формул могут соответствовать выбранной элементарной формуле, и машина вывода должна выбрать одну из них, чтобы попытаться сделать унификацию. Правило поиска в языке Prolog определяет, что формулы выбираются в том порядке, в котором они появляются в тексте программы. При попытке установить соответствие целевой формулы с формулами процедуры substring правило поиска требует, чтобы сначала была выбрана истинная substring (T,T), затем вторая формула с substring (S, T1), и, только если она не выполняется, третья формула с substring (S, T2). ,

Основанием для этих, по-видимому, произвольных требований, послужило то, что они дают возможность реализовать язык Prolog на стековой архитектуре точно так же, как языки С и Ada, и сделать большинство вычислений в языке Prolog столь же эффективными, как и в процедурных языках. Вычисление выполняется *перебором с откатами (backtracking).* В приведенном выше примере:

?- concat ("Hello world", T1, T2), substring ("wor", T1).

предположим, что вычисление выбрало для concat подстановку

["H" -»tl, "ello world" -> t2]

Теперь делается попытка доказать substring ("wor", "H"), которая, очевидно, не выполняется. Вычисление делает откат и пытается найти другое доказательство concat с другой подстановкой. Все данные, необходимые для вычисления substring ("wor", "H"), можно отбросить после отката. Таким образом, правило вычисления в языке Prolog естественно и эффективно реализуется на стеке.

Чтобы еще улучшить эффективность программ, написанных на языке Prolog, в язык включена возможность, названная *обрезанием (cut* обозначается «!»), которая позволяет задать указание машине вывода воздержаться от поиска части пространства возможных решений. Именно программист должен гарантировать, что никакие возможные решения не «обрезаны». Например, предположим, что мы пытаемся проанализировать арифметическое выражение, которое определено как два терма, разделенных знаком операции:

expression (T1, OP, T2):- term (T1), operator (OP), !, term (T2).

operator ('+').
operator ('-').
operator ('*').
operator ('/').

и что цель — expression (n, '+', 27). Очевидно, что и n и 27 являются термами, а '+' — одним из операторов, поэтому цель выполняется. Если, однако, в качестве цели задать expression (n,'+', '>'), то вычисление при отсутствии обрезания продолжится следующим образом:

n — терм
'+' соответствует operator ('+')
'>' —нетерм
'+' не соответствует operator('-')
'+' не соответствует operator ('*')
'+' не соответствует operator ('/')

Машина вывода делает откат и пытается доказать operator (OP) другими способами в надежде, что другое соответствие позволит также доказать term (T2). Конечно, программист знает, что это безнадежно: обрезание приводит к тому, что вся формула для expression дает неуспех, если неуспех происходит после того, как будет пройдено обрезание. Конечно, обрезание уводит язык Prolog еще дальше от идеального декларативного логического программирования, но обрезание активно используют на практике для улучшения эффективности программы.

Нелогические формулы

Для практического применения в язык Prolog включены свойства, которые не имеют никакого отношения к логическому программированию. По определению операторы вывода не имеют никакого логического значения в вычислении, поскольку их результат относится только к некоторой среде за пределами программы. Однако операторы вывода необходимы при написании программ, которые открывают файлы, отображают символы на экране и т. п.

Другая область, в которой язык Prolog отходит от чистого логического программирования, — численные вычисления. Конечно, в логике можно определить сложение; фактически, это единственный способ определить сложение строго:

N + 0 = N
N + s (M) = s (K) <= N + M = K

O — это числовой ноль, а s(N) — выражение для числа, следующего за N, так, например, s(s(s(0))) — выражение для числа 3. Формулы определяют «+», используя два правила: 1) число плюс ноль — это само число, и 2) N плюс следующее за M — это следующее за N + M. Очевидно, было бы чрезвычайно утомительно писать и неэффективно выполнять логическую версию для 555 + 777.

Prolog включает элементарную формулу:

Var is Expression

Вычисляется значение Expression, и создается новая переменная Var с этим значением. Обратите внимание, что это не присваивание; переменной нельзя присвоить значение еще раз, ее только можно будет использовать позднее как аргумент в какой-нибудь элементарной формуле.

Вычисление выражения и присваивание вновь созданной переменной можно использовать для моделирования циклов:

```
loop (0).
loop (N) :-
   proc,
   N1 isN-1,
   loop(N1).
```

Следующая цель выполнит proc десять раз:

```
?-loop (10).
```

Аргумент является переменной, которая используется как индекс. Первая формула — базовый случай рекурсии: когда индекс равен нулю, больше ничего делать не нужно. В противном случае выполняется процедура proc, создается *новая* переменная N1 со значениями N-1, которая используется как аргумент для рекурсивного вызова loop. Унификация создает новую переменную для каждого использования второй формулы loop. Нельзя сказать, что это слишком неэффективно, потому что это можно выполнить в стеке. К тому же многие компиляторы Prolog могут делать оптимизацию хвостовой рекурсии, т. е. заменять рекурсию обычной итерацией, если рекурсивный вызов является последним оператором в процедуре.
Причина того, что использование is — нелогическое, состоит в том, что оно не симметрично, т. е. вы не можете написать:

```
28 is V1 * V2
```

или даже:

```
28 is V1*7
```

где V1 и V2 — переменные, которые еще не получили своих значений. Для этого потребовалось бы знать семантику арифметики (как разлагать на множите ли и делить целые числа), в то время как унификация устанавливает только синтаксическое соответствие термов.

База данных на языке Prolog

Нет никаких других ограничений на количество формул, которые может содержать программа на языке Prolog, кроме ограничений, связанных с размером памяти компьютера. В частности, нет ограничения на количество фактов, которые можно включить в программу, поэтому набор фактов на языке Prolog может выполнять функцию таблицы в базе данных:

```
customer(1, "Jonathan").                    /* клиент(Идент_клиента, Имя) */
customer(2, "Marilyn"),
customer^, "Robert").

salesperson 101, "Sharon").                 /* продавец(Идент_продавца, Имя) */
salesperson 102, "Betty").
salesperson 103, "Martin").

order(103, 3, "Jaguar").                    /*заказ(Идент_продавца,
order(101, 1, "Volvo").                           Идент_клиента, товар)*/
order(102, 2, "Volvo").
```

```
order(103, 1, "Buick").
```

Обычные цели языка Prolog могут интерпретироваться как запросы к базе данных. Например:

```
?-   salesperson(SalesJD, "Sharon"),          /* ID Шэрон */
order(SalesJD, CustJD, "Volvo"),              /* Заказ Volvo */
customer(CustJD, Name).                       /* Клиент заказа */
```

означает следующее: «Кому Шэрон продала *Volvo?*». Если запрос успешный, переменная Name получит значение имени одного из клиентов. В противном случае мы можем заключить, что Шэрон никому *Volvo* не продавала.

Сложные запросы базы данных становятся простыми целями в языке Prolog. Например: «Есть ли клиент, которому продавали автомобиль и Шэрон, и Мартин?»:

```
?-    salesperson(ID1,"Sharon"),          /* ID Шэрон */
      salesperson(ID2, "Martin"),         /* ID Мартина */
      order(ID1, CustJD, _),              /* ID клиента Шэрон */
     order(ID2, CustJD, _).               /* ID клиента Мартина */
```

Поскольку переменная CustJD является общей для двух элементарных формул, цель может быть истинной только, когда один и тот же клиент делал заказ у каждого из продавцов.

Является ли язык Prolog реальной альтернативой специализированному программному обеспечению баз данных? Реализация списков фактов вполне эффективна и может легко отвечать на запросы для таблиц из тысяч записей. Однако, если ваши таблицы содержат десятки тысяч записей, необходимы более сложные алгоритмы поиска. К тому же, если ваша база данных предназначена для непрограммистов, необходим соответствующий интерфейс пользователя, и в этом случае ваша реализация языка Prolog может и не оказаться подходящим языком программирования.

Важно подчеркнуть, что «это не было сделано профессионалами», т. е. мы не вводили ни новый язык, ни понятия базы данных; это было всего лишь обычное программирование на языке Prolog. Любой программист может создавать небольшие базы данных, просто перечисляя факты, а затем в любой момент выдавать запросы.

Динамические базы данных

Если все наборы фактов, существуют с самого начала программы на языке Prolog, запросы совершенно декларативны: они только просят о заключении, основанном на ряде предположений (фактов). Однако язык Prolog включает нелогическое средство, с помощью-которого можно менять базу данных в процессе вывода. Элементарная формула assert(F) всегда истинна как логическая формула, но в качестве побочного эффекта она добавляет факт F к базе данных; точно так же retract(F) удаляет факт F:

```
?-    assert(order( 102, 2, "Peugeot")),        /* Бетти продает автомобиль'*/
      assert(order(103,1 , "BMW")),             /* Мартин продает автомобиль */
      assert(order(102, 1, "Toyota")),          /* Бетти продает автомобиль*/
      assert(order(102, 3, "Fiat")),            /* Бетти продает автомобиль */
      retract(salesperson(101, "Sharon")).      /* Уволить Шэрон! */
```

С помощью изменений базы данных можно в языке Prolog смоделировать оператор присваивания. Предположим, что факт count(O) существует в программе, тогда:

```
increment :-
    N1 is N +1,                 /* Новая переменная с новым значением */
    retract(count(N)),          /* Стереть старое значение */
    assert(count(N1)).          /* Сохранить новое значение */
```

Ни одна из трех элементарных формул не является логической!

Вспомните, что присваивание используется, чтобы записать состояние вычисления. Таким образом, альтернатива моделированию присваивания — внести каждую переменную состояния как дополнительный аргумент в формулы, которые могут стать и сложными, и запутанными. На практике в программах на языке Prolog допустимо использовать нелогические операции с базой данных как для того, чтобы реализовать динамические базы данных, так и для того, чтобы улучшить читаемость программы.

Сортировка в языке Prolog

В качестве примера соотношения между описательным и процедурным взглядами на логическую программу мы обсудим программы сортировки на языке Prolog. Мы ограничимся сортировкой списков целых чисел. Обозначения: [Head]Tail] является списком, первый элемент которого — Head, а остальные элементы образуют список Tail. [] обозначает пустой список.

Сортировка в логическом программировании вполне тривиальна, потому нам нужно только описать смысл того, что список L2 является отсортированной версией списка L1. Это означает, что L*2* представляет собой перестановку (permutation) всех элементов L1 при условии, что элементы упорядочены (ordered):

```
sort(L1, L2):- permutation(L1, L2), ordered(L2).
```

где формулы в теле процедуры определены как:

```
permutation([], []).
permutation(L, [X | Tail]) :-
      append(Left_Part, [X | Right_Part], L),
       append(Left_Part, Right_Part, ShortJJst),
        permutation(Short__List, Tail).

ordered([]).
ordered([Single]).
ordered([First, Second | Tail]) :-
       First =< Second,
       ordered([Second | Tail]).
```

Прочитаем их описательно:

• Пустой список является перестановкой пустого списка. Перестановка непустого списка является разделением списка на элемент X и две части Left_Part и Right_Part, так, что X добавляется в начало перестановки конкатенации двух частей. Например:
permutation([7,2,9,3], [91Tail])
если Tail является перестановкой [7,2,3].

• Список с не более чем одним элементом упорядочен. Список упорядочен, если первые два элемента упорядочены, и список, состоящий из всех элементов, кроме первого, также упорядочен.

С точки зрения процедуры это не самая эффективная программа сортировки; действительно, ее обычно называют *медленной сортировкой! Она* просто перебирает (генерирует) все перестановки списка чисел, пока не найдет отсортированный список. Однако также просто написать описательную версию более эффективных алгоритмов сортировки типа *сортировки вставкой,* который мы рассмотрели на языке ML в предыдущей главе:

```
insertion_sort([], []).
insertion_sort([Head | Tail], List) :-
        insertion_sort(Tail, Tail _1),
        insert_element(Head, Tail_1, List).

insert_element(X, [], [X]).
insert_element(X, [Head | Tail], [X, Head | Tail]) :-
         X=<Head.
insert_element(X, [Head   Tail], [Head   Tail_1]) :-
         insert_element(X, Tail, Tail_1).
```

С процедурной точки зрения программа вполне эффективна, потому что она выполняет сортировку, непосредственно манипулируя подсписками, избегая бесцельного поиска. Как и в функциональном программировании, здесь нет никаких индексов, циклов и явных указателей, и алгоритм легко обобщается для сортировки других объектов.

Типизация и «неуспех»

В языке Prolog нет статического контроля соответствия типов. К сожалению, реакция машины вывода языка Prolog на ошибки, связанные с типом переменных, может вызывать серьезные затруднения для программиста. Предположим, что мы пишем процедуру для вычисления длины списка:

```
length([], 0).                    /* Длина пустого списка равна 0 */
length([Head | Tail], N) :      - /* Длина списка равна */
length(Tail, N1),                 /* длине Tail */
N is N1+1.                        /* плюс 1*/
```

и случайно вызываем ее с целочисленным значением вместо списка:

```
?- length(5, Len).
```

Это не запрещено, потому что вполне возможно, что определение length содержит дополнительную нужную для отождествления формулу.

Машина вывода при вызове lenght в качестве реакции просто даст неуспех, что совсем не то, что вы ожидали. А length была вызвана внутри некоторой другой формулы p, и неуспех length приведет к.неуспеху p (чего вы также не ожидали), и так далее назад по цепочке вызовов. Результатом будут неуправляемые откаты, которые в конце концов приведут к неуспеху первоначальной цели при отсутствии какой-либо очевидной причины. Поиск таких ошибок — очень трудный процесс трассировки вызовов шаг за шагом, пока ошибка не будет диагностирована.

По этой причине некоторые диалекты языка Prolog типизированы и требуют, чтобы вы объявили, что аргумент является или целым числом, или списком, или некоторым типом, определенным программистом. В типизированном языке Prolog вышеупомянутый вызов был бы ошибкой компиляции. В таких диалектах мы снова встречаем привычный компромисс: обнаружение ошибок во время компиляции за счет меньшей гибкости.

17.4. Более сложные понятия логического программирования

Успех языка Prolog стимулировал появление других языков логического программирования. Многие языки попытались объединить преимущества логического программирования с другими парадигмами программирования типа объектно-ориентированного и функционального программирования. Вероятно, наибольшие усилия были вложены в попытки использовать параллелизм, свойственный логическому программированию. Вспомните, что логическая программа состоит из последовательности формул:

t=>t
(t = tl || t2)^(SCtl)=>(S=>t)
(t = tl || 12) ^ (s c t2) => (s => t)

Язык Prolog вычисляет каждую цель последовательно слева направо, но цели можно вычислять и одновременно. Это называется *и-параллелизмом* из-за конъюнкции, которая соединяет формулы цели. Сопоставляя цели с головами формул программы, язык Prolog проверяет каждую формулу последовательно в том порядке, в котором она встречается в тексте, но можно проверять формулы и одновременно. Это называется *или-параллелизмом*, потому что каждая цель должна соответствовать первой *или* второй формуле, и т. д.

При создании параллельных логических языков возникают трудности. Проблема м-параллелизма — это проблема синхронизации: когда одна переменная появляется в двух различных целях, подобно tl в примере, фактически только одна цель может конкретизировать переменную (записывать в нее), и это должно заблокировать другие цели от чтения переменной прежде, чем будет выполнена запись. В мям-параллелизме несколько процессов выполняют параллельный поиск решения, один для каждой формулы процедуры; когда решение найдено, должны быть выполнены некоторые действия, чтобы сообщить этот факт другим процессам, и они могли бы завершить свои поиски.

Много усилий также прилагалось для интеграции функционального и логического программирования. Существует очень близкая связь между математикой функций и логикой, потому что:

$y = f(x_b ...,x_{„})$

Основные различия между двумя концепциями программирования следующие:

1. Логическое программирование использует (двунаправленную) унификацию, что сильнее (однонаправленного) сопоставления с образцом, используемого в функциональном программировании.

2. Функциональные программы являются однонаправленными в том смысле, что, получив все аргументы, программа возвращает значение. В логических программах любой из аргументов цели может остаться неопределенным, и ответственность за его конкретизацию (instantiating) в соответствии с ответом лежит на унификации.

3. Логическое программирование базируется на машине вывода, которая автоматически ищет ответы.

4. Функциональное программирование оперирует с объектами более высокого уровня абстракции, поскольку функции и типы можно использовать и как данные, в то время как логическое программирование более или менее ограничено формулами на обычных типах данных.

5. Точно так же средства высокого порядка в функциональных языках программирования естественно обобщаются на модули, в то время как логические языки программирования обычно «неструктурированы».

Новая область исследования в логическом программировании — расширение отождествления от простой синтаксической унификации к включению семантической информации. Например, если цель определяет 4 < x < 8 и голова формулы определяет 6 < x < 10, то мы можем заключить, что 6 s x < 8 и что x = 6 или x = 7. Языки, которые включают семантическую информацию при отождествлении, называются *ограниченными (constraint) логическими языками программирования,* потому что значения ограничиваются уравнениями. Ограниченные логические языки программирования должны базироваться на эффективных алгоритмах для решения уравнений.

Продвижение в этом направлении открывает новые перспективы повышения как уровня абстракции, так и эффективности логических языков программирования.

17.5. Упражнения

1. Вычислите 3 + 4, используя логическое определение сложения.

2. Что произойдет, если программа loop вызвана с целью 1oop(-1)? Как можно исправить программу?

3. Напишите программу, которая не завершается из-за конкретного правила вычисления, принятого в языке Prolog . Напишите программу, которая не завершается из-за правила поиска.

4. По правилам языка Prolog поиск решений осуществляется *сначала вглубь (depth-first search),* поскольку крайняя левая формула неоднократно выбирается даже после того, как она была заменена. Также можно делать поиск *сначала в ширину (breath-first search),* выбирая формулы последовательно слева направо и возвращаясь к крайней левой формуле, только когда все другие уже выбраны. Как влияет это правило на успех вычисления?

5. Напишите цель на языке Prolog для следующего запроса:
Есть ли тип автомобиля, который был продан Шэрон, но не Бетти?
Если да, какой это был автомобиль?

6. Изучите встроенную формулу языка Prolog findall и покажите, как она может ответить на следующие запросы:
Сколько автомобилей продал Мартин?
Продал ли Мартин больше автомобилей, чем Шэрон?

7. Напишите программу на языке Prolog для конкатенации списков и сравните ее с программой на языке ML. Выполните программу на языке Prolog в различных «направлениях».

8. Как можно изменить процедуру языка Prolog, чтобы она улавливала несоответствия типов и выдавала сообщение об ошибке?

Какие типы логических программ извлекли бы выгоду из м-параллелиз-ма, а какие — из м/ш-параллелизма?

Глава 18

Java

Java является и языком программирования, и моделью для разработки программ для сетей. Мы начнем обсуждение с описания модели Java, чтобы показать, что модель не зависит от языка. Язык интересен сам по себе, но повышенный интерес к Java вызван больше моделью, чем свойствами языка.

18.1. Модель Java

Можно написать компилятор для языка Java точно так же, как и для любого другого процедурного объектно-ориентированного языка. Модель Java, однако, базируется на концепции интерпретатора (см. рис. 3.2, приведенный в измененной форме как рис. 18.1), подобного интерпретаторам для языка Pascal, которые мы обсуждали в разделе 3.10.

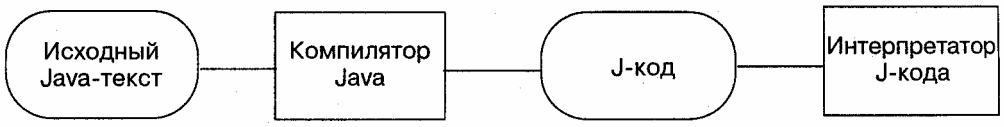

Рис. 18.1. Интерпретатор для Java.

В Java стрелка от J-кода к интерпретатору J-кода представляет не просто поток данных между компонентами среды разработки программного обеспечения. Вместо этого J-код может быть упакован в так называемый *аплет (applet),* который может быть передан по компьютерной системе сети. Принимающий компьютер выполняет J-код, используя интерпретатор, называющийся *виртуальной Java машиной (Java Virtual Machine — JVM).* JVM обычно встроена внутрь *браузера* сети, который является программой поиска и отображения информации, получаемой по сети. Когда браузер определяет, что был получен аплет, он вызывает JVM, чтобы выполнить J-код. Кроме того, модель Java включает стандартные библиотеки для графических интерфейсов пользователя, мультимедиа и сетевой связи, которые отображаются каждой реализацией JVM на средства основной операционной системы. Это является значительным расширением концепции виртуальной машины по сравнению с простым вычислением P-кода для языка Pascal.
Обратите внимание, что также можно написать обычную программу на Java, которая называется *приложением.* Приложение не подчиняется ограничениям по защите данных, рассматриваемым ниже. К сожалению, имеются несколько огорчающих различий между программированием аплета и приложения.

Проблема эффективности

Слишком хорошо, чтобы быть правдой? Конечно. Модель Java страдает от тех же самых проблем эффективности, которые присущи любой модели, основанной на интерпретации абстрактного машинного кода. Для относительно простых программ, выполняющихся на мощных персональных компьютерах и рабочих станциях, это не вызовет серьезных проблем, но производительность может установить ограничение на применимость модели Java.

Одно из решений состоит в том, чтобы *включить в браузер* на получающей стороне компилятор, который переводит абстрактный J-код в машинный код принимающего компьютера. Фактически, компьютер может выполнять этот перевод одновременно или почти одновременно с приемом J-кода; это называется *компиляцией «налету»*. В худшем случае скорость работы возрастет при *втором* выполнении аплета, который вы загрузили.

Однако это решение — только частичное. Не очень практично встраивать сложный оптимизирующий компилятор внутрь каждого браузера для каждой комбинации компьютера и операционной системы. Тем не менее, для многих типов прикладных программ модель Java вполне подойдет для разработки переносимого программного обеспечения.

Проблема безопасности

Предположим, что вы загружаете аплет и выполняете его на своем компьютере. Откуда вы знаете, что он не собирается затереть ваш жесткий диск? Оценив существенный ущерб от компьютерных вирусов, злонамеренно внедренных во вполне хорошие в других отношениях программы, вы можете понять, как опасно загружать произвольные программы, взятые с удаленного пункта сети. Модель Java использует несколько стратегий устранения (или по крайней мере уменьшения!) возможности того, что аплет, пришедший по сети, повредит программное обеспечение и данные на принявшем его компьютере:

• **Строгий контроль соответствия типов** как в языке Ada. Идея состоит в том, чтобы устранить случайное или преднамеренное повреждение данных, вызванное выходом индексов за границы массива, произвольными или повисшими указателями и т. д. Мы рассмотрим этот вопрос подробно в следующих разделах.

• **Проверка J-кода.** Интерпретатор на принимающем компьютере проверяет, действительно ли поток байтов, полученных с удаленного компьютера, состоит из допустимых инструкций J-кода. Это гарантирует, что реально выполняется семантика безопасности модели и что не удастся «обмануть» интерпретатор и причинить вред.

• **Ограничения на аплет.** Аплету не разрешается выполнять некоторые операции на получающем компьютере, например запись или удаление файлов. Это наиболее проблематичный аспект модели безопасности, потому что хотелось бы писать аплеты, которые могут делать все, что может делать обычная программа.

Ясно, что успех Java зависит от того, является ли модель безопасности достаточно строгой, чтобы предотвратить злонамеренное использование JVM, в то же самое время сохраняя достаточно возможностей для создания полезных программ.

Независимость модели от языка

Проницательный читатель может заметить, что в предыдущем разделе не было ссылок на язык программирования Java! Это сделано специально, потому что модель Java является и эффективной, и полезной, даже если исходный текст аплетов написан на каком-нибудь другом языке. Например, существуют компиляторы, которые переводят Ada95 в J-Код[3]. Однако язык Java был разработан вместе с JVM, и семантика языка почти полностью соответствует возможностям модели.

18.2. Язык Java

На первый взгляд синтаксис и семантика языка Java аналогичны принятым в языке C++. Однако в то время как C++ сохраняет почти полную совместимость с языком C, Java отказывается от совместимости ради устранения трудностей, связанных с проблематичными конструкциями языка C. Несмотря на внешнее сходство, языки Java и C++ весьма различны, и программу на C++ не так легко перенести на Java.
В основном языки похожи в следующих областях:

• Элементарные типы данных, выражения и управляющие операторы.

• Функции и параметры.

• Объявления класса, члены класса и достижимость.

• Наследование и динамический полиморфизм.

• Исключения.

В следующих разделах обсуждается пять областей, где проект Java существенно отличается от C++: семантика ссылки, полиморфные структуры данных, инкапсуляция, параллелизм и библиотеки. В упражнениях мы просим вас изучить другие различия между языками.

18.3. Семантика ссылки

Возможно, наихудшее свойство языка C (и C++) — неограниченное и чрезмерное использование указателей. Причем операции с указателями не только трудны для понимания, они чрезвычайно предрасположены к ошибкам, как описано в гл. 8. Ситуация в языке Ada намного лучше, потому что строгий контроль соответствия типов и уровни доступа гарантируют, что использование указателей не приведет к разрушению системы типов, однако структуры данных по-прежнему должны формироваться с помощью указателей.

Язык Java (подобно Eifiel и Smalltalk) использует *семантику ссылки* вместо *семантики значения.*

При объявлении переменной непримитивного типа память не выделяется; вместо этого выделяется неявный указатель. Чтобы реально выделить память для переменной, необходим второй шаг. Покажем теперь, как семантика ссылки работает в языке Java.

Массивы

Если вы объявляете массив в языке С, то выделяется память, которую вы запросили (см. рис. 18.2а):

```
inta_c[10];
```
C

в то время как в языке Java вы получаете только указатель, который может использоваться для обращений к массиву (см. рис. 18.26):

```
int[ ] a Java;
```
Java

Для размещения массива требуется дополнительная команда (см. рис. 18.2 в):

```
a Java = new int[10];
```
Java

хотя допустимо объединять объявления с запросом памяти:

```
int[ ] a Java = new int[10];
```
Java

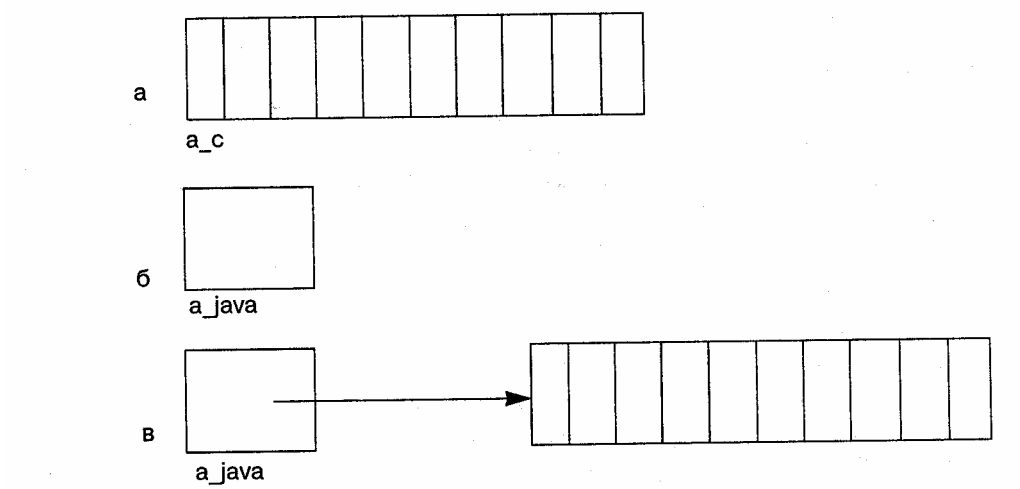

Рис. 18.2. Семантика значения в сравнении с семантикой ссылки.

Если вы сравните рис. 18.2 с рис. 8.4, вы увидите, что массивы в Java скорее подобны структурам, определенным в языке C++ как int *a, а не как int a []. Различие заключается в том, что указатель является неявным, поэтому вы не должны заботиться об операциях с указателями или о выделении памяти. К тому же, в случае массива, переменная будет дескриптором массива (см. рис. 5.4), что дает возможность проверять границы при обращениях к массиву.

Отметим, что синтаксис Java проще читается, чем синтаксис C++: a_java is of type int [], что означает «целочисленный массив»; в языке C++ тип компонентов int и указание «масивности» [10] располагаются по разные стороны от имени переменной.

При использовании семантики ссылки разыменование указателя является неявным, поэтому после того, как массив создан, вы обращаетесь к нему как обычно:

```
for (i = 1; i< 10;i++)
```
Java

```
a_java[i] = i;
```

Конечно, косвенный доступ может быть значительно менее эффективен, чем прямой, если его не оптимизирует компилятор.

Отметим, что выделение памяти для объекта и присваивание его переменной могут быть выполнены в любом операторе, в результате чего появляется следующая возможность:

```
int[ ] a_ Java = new int[10];
…
a_Java = new int[20];
```
Java

Переменная a_ Java, которая указывала на массив из десяти элементов, теперь указывает на массив из двадцати элементов, и исходный массив становится «мусором» (см. рис. 8.7). Согласно модели Java сборщик мусора должен находиться внутри JVM.

Динамические структуры данных

Как можно создавать списки и деревья без указателей?! Объявления для связанных списков в языках C++ и Ada, описанные в разделе 8.2, казалось бы, показывали, что нам нужен указательный тип для описания типа следующего поля next:

```
typedef struct node *Ptr;
typedef struct node {
    int data;
    Ptr next;
} node;
```
C

Но в Java каждый объект непримитивного типа автоматически является указателем

```
class Node {
    int data;
    Node next;
}
```
Java

Поле next — это просто указатель на узел, а не сам узел, поэтому в объявлении нет никакой цикличности. Объявление списка — это просто:

```
Node head;
```
Java

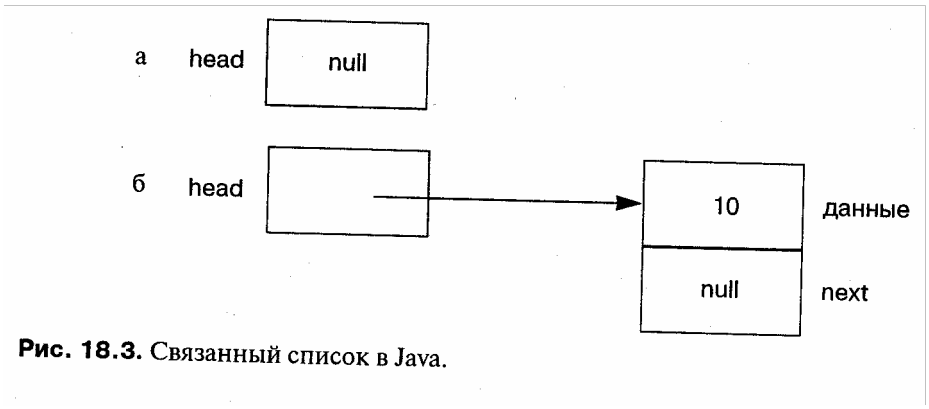

Рис. 18.3. Связанный список в Java.

Этот оператор создает указатель переменной с нулевым значением (см. рис. 18.3,а). Подразумевая, что имеется соответствующий конструктор (см. раздел 15.3) для Node, следующий оператор создает узел в головной части списка (см. рис. 18.3,б):

```
head = new Node(10, head);
```
Java

Проверка равенства и присваивание

Поведение операторов присваивания и проверки равенства в языках с семантикой ссылки может оказаться неожиданным для программистов, которые работали на языке с семантикой значения. Рассмотрим объявления Java:

```
String s1 = new String("Hello");
 String s2 = new String("Hello");
```
Java

В результате получается структура данных, показанная на рис. 18.4. Теперь предположим, что мы сравниваем строковые переменные:

```
if (s1 == s2) System.out.println("Equai");
else System.out.println("Not equal");
```
Java

программа напечатает Not equal (Не равно)! Причина этого хорошо видна из рис. 18.4: переменные являются указателями с разными значениями, и тот факт, что они указывают на равные массивы, не имеет значения. Точно так же, если мы присваиваем одну строку другой s1 = s2, будут присвоены указатели, но никакого копирования значений при этом не будет. В этом случае, конечно, s1 == s2 будет истинно. Java делает различие между *мелкими* копированием и проверкой равенства и *глубокими* копированием и сравнением. Последние объявлены в классе Object — общем классе-прародителе — и названы clone и equals. Предопределенный класс String, например, переопределяет эти операции, поэтому s1.equals(s2) будет истинно. Вы можете также переопределить эти операции, чтобы создать глубокие операции для своих классов.

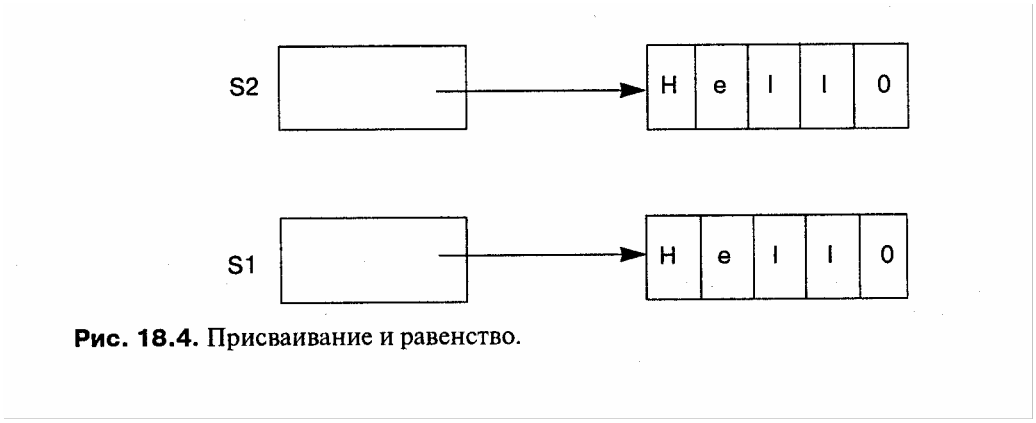

Рис. 18.4. Присваивание и равенство.

Подведем итог использования семантики ссылки в Java:

• Можно безопасно управлять гибкими структурами данных.

• Программирование упрощается, потому что не нужны явные указатели.

• Есть определенные издержки, связанные с косвенностью доступа к структурам данных.

18.4. Полиморфные структуры данных

В языках Ada и C++ есть два пути построения полиморфных структур данных: generics в Ada и templates в C++ для полиморфизма на этапе компиляции, и типы в Ada и указатели/ссылки на классы для полиморфизма на CW-этапе выполнения. Преимущество generies/templates состоит в том, что структура данных фиксируется при создании экземпляра во время компиляции; это позволяет как генерировать более эффективный код, так и более экономно распределять память для структур данных.

В языке Java решено реализовать полиморфизм только на этапе выполнения. Как и в языках Smalltalk и Eiffel, считается, что каждый класс в Java порождается из корневого класса, названного Object. Это означает, что значение любого непримитивного типа[8] может быть присвоено объекту типа Object. (Конечно, это работает благодаря семантике ссылки.)

Чтобы создать связанный список, класс Node должен быть сначала определен как содержащий (указатель на) Object. Класс списка тогда должен содержать методы вставки и поиска значения типа Object:

```
class Node {
    Object data;
     Node next;
}
 class List {
       private Node head;
       void Put(Object data) {...};
       Object Get() {...};
}
```

Java

Java

Если L является объектом типа List (Список), и a является объектом типа Airplane_Data, то допустимо L.Put(a), потому что Airplane_Data порождено из Object. Когда значение выбирается из списка, оно должно быть приведено к соответствующему потомку Object:

```
a = (Airplane_Data) List.Get();
```

Java

Конечно, если возвращенное значение не имеет тип Airplane_Data (или не порождено из этого типа), возникнет исключение.

Преимущество этой парадигмы состоит в том, что в Java очень просто писать обобщенные структуры данных, но по сравнению с generics/tem-plates имеются два недостатка: 1) дополнительные издержки семантики ссылки (даже для списка целых чисел!), и 2) опасность, что объект, размещенный не в той очереди, приведет при поиске к ошибке на этапе выполнения программы.

18.5. Инкапсуляция

В разделе 13.1 мы обсуждали тот факт, что в языке C нет специальной конструкции инкапсуляции, а в разделе 13.5 отметили, что операция разрешения области действия и конструкция namespace (пространство имен) в C++ уточняет грубое приближение языка C к проблеме видимости глобальных имен. Для совместимости в язык C++ также не была включена конструкция инкапсуляции; вместо этого сохраняется зависимость от «h»-файлов. В Ada есть конструкция пакета, которая поддерживает инкапсуляцию конструкций в модули (см. раздел 13.3), причем спецификации пакетов и их реализации (тела) могут компилироваться отдельно. Конструкции with позволяют разработчику программного обеспечения точно определить зависимости между модулями и использовать порожденные пакеты (кратко упомянутые в разделе 15.2) для разработки модульных структур с иерархической достижимостью.

Java содержит конструкцию инкапсуляции, названную *пакетом (package)*, но, к сожалению, конструкция эта по духу ближе к пространству имен (namespace) в языке C++, чем к пакету Ada! Пакет является совокупностью классов:

```
package Airplane_Package;
  public class Airplane_Data
 {
        int speed;                                  // Доступно в пакете
       private int mach_speed;                      // Доступно в  классе
        public void set_speed(int s) {...};         // Глобально доступно
        public int get_speed() {...};
}
public class Airplane_Database
{
   public void new_airplane(Airplane_Data a, int i)
  {
    if (a.speed > 1000) // Верно !
       a.speed = a.mach_speed; // Ошибка !
   }
private Airplane_Data[ ] database = new Airplane_Data[1000];
}
```

Java

Пакет может быть разделен на несколько файлов, но файл может содержать классы только из одного пакета.

Спецификаторы public и private аналогичны принятым в языке C++: public (общий) означает, что элемент доступен за пределами класса, в то время как private (приватный) ограничивает достижимость для других членов класса. Если никакой спецификатор не задан, то элемент видим *внутри* пакета. В примере мы видим, что элемент int speed (скорость) класса Airplane_Data не имеет никакого спецификатора, поэтому к нему может обратиться оператор внутри класса Airplane_Database, так как два класса находятся в одном и том же пакете. Элемент mach_speed объявлен как private, поэтому он доступен только внутри класса Airplane_Data, в котором он объявлен.

Точно так же классы имеют спецификаторы достижимости. В примере оба класса объявлены как public, что означает, что другие пакеты могут обращаться к любому (public) элементу этих классов. Если класс объявлен как private, он доступен только внутри пакета. Например, мы могли бы объявить private класс Airplane_File, который использовался бы внутри пакета для записи в базу данных.

Пакеты играют важную роль в развитии программного обеспечения Java, потому что они позволяют группировать вместе связанные классы при сохранении явного контроля над внешним интерфейсом. Иерархическая библиотечная структура упрощает построение программных инструментальных средств.

Сравнение с другими языками

Пакеты Java служат для управления глобальными именами и достижимостью аналогично конструкции namespace в языке C++. При заданных в нашем примере объявлениях любая Java-программа может содержать:

```
Airplane_Package.Airplane_Data a;
a.set_speed(100);
```

Java

потому что имена класса и метода объявлены как public. Не изучив полный исходный текст пакета, нельзя узнать, какие именно классы импортированы. Есть оператор import, который

открывает пространство имен пакета, разрешая прямую видимость. Эта конструкция аналогична конструкциям using в C++ и uses Ada.

Основное различие между Java и Ada состоит в том, что в Ada спецификация пакета и тело пакета разделены. Это не только удобно для сокращения размера компиляций, но и является существенным фактором в разработке и поддержке больших программных систем. Спецификация пакета может быть заморожена, позволяя параллельно разрабатывать тело пакета и вести разработку других частей. В Java «интерфейс» пакета является просто совокупностью всех public-объявлений. Разработка больших систем на Java требует, чтобы программные инструментальные средства извлекали спецификации пакета и гарантировали совместимость спецификации и реализации.

Конструкция пакета дает Java одно главное преимущество перед C++. Пакеты сами используют соглашение об иерархических именах, которое позволяет компилятору и интерпретатору автоматически размещать классы. Например, стандартная библиотека содержит функцию, названную Java.lang.String.toUpperCase. Это интерпретируется точно так же, как операционная система интерпретирует расширенное имя файла: toUpperCase является функцией в пакете Java.lang.String. Библиотеки Java могут (но не обязаны) быть реализованы как иерархические каталоги, где каждая функция является файлом в каталоге для своего класса. Отметим, что иерархичность имен как бы *вне* языка; подпакет не имеет никаких особых привилегий при доступе к своему родителю. Здесь мы видим четкое отличие от пакетов-детей в Ada, которые имеют доступ к private-декларациям своих родителей при соблюдении правил, которые не позволяют экспортировать эти декларации.

18.6. Параллелизм

Ada — один из немногих языков, в которых поддержка параллелизма включена в сам язык, в отличие от делегирования этих функций операционной системе. Язык Java продолжает идеологию языка Ada в отношении переносимости параллельных программ вне зависимости от операционных систем. Важное применение параллелизма в Java — программирование серверов: каждый запрос клиента заставляет сервер *порождать (spawn)* новый процесс для выполнения этого запроса.

Параллельно исполняемые конструкции Java называются *нитями (thread).* Собственно в параллелизме нет никаких существенных различий между нитью и тем, что называют стандартным термином *процесс;* отличие состоит в реализации, ориентированной на выполнение многих нитей внутри одного и того же адресного пространства. Для разработки и анализа параллельных алгоритмов используется та же самая модель, что и в гл. 12 ,— чередующееся выполнение атомарных инструкций процессов.

Класс Java, который наследуется из класса Thread, объявляет только новый тип *нить*. Чтобы реально создать нить, нужно запросить память, а затем вызвать функцию start. В результате будет запущен метод run внутри нити:

```
class My_Thread extends Thread
{
    public void run(){...};                    // Вызывается функцией start
}                                                                          Java
My_Thread t = new My_Thread();          // Создать нить
 t.start();                                    //Активизировать нить
```

Одна нить может явно создавать, уничтожать, блокировать и разблокировать другую нить. Эти конструкции аналогичны конструкциям в языке Ada, которые позволяют определить тип task (задача) и затем динамически создавать задачи.

Синхронизация

Java поддерживает форму синхронизации аналогично мониторам (см. раздел 12.4). Класс может содержать методы, специфицированные как synchronized (синхронный). С каждым таким объектом в классе связывается блокировка-пропускник (lock), которая гарантирует, что только одна нить в данный момент может выполнять синхронный метод в объекте. Следующий пример показывает, как объявить монитор для защиты разделяемого ресурса от одновременного использования несколькими нитями:

```
class Monitor
{
   synchronized public void seize() throws InterruptedException
{
     while (busy) wait();
      busy = true;
}
synchronized public void release()
{
     busy = false;
     notify();
 }
     private boolean busy - false
}
```

Java

Монитор использует булеву переменную, которая указывает состояние ресурса. Если две нити пытаются выполнить метод seize в мониторе, то только одна из них пройдет пропускник и выполнится. Эта нить установит переменную busy (занято) в состояние true (истина) и перейдет к использованию ресурса. По завершении метода пропускник откроется, и другая нить сможет выполнить метод seize. Теперь, однако, переменная busy будет иметь значение false (ложь). Вместо ненужных затрат времени ЦП на непрерывную проверку переменной нить предпочитает освободить ЦП с помощью запроса wait (ждать). Когда первая нить заканчивает использование разделяемого ресурса, она вызывает notify (уведомление), которое позволит ожидающей нити снова возобновить выполнение синхронного метода.

Конструкции Java для параллелизма достаточно просты. Нет ничего похожего на сложные рандеву Ada для прямой связи процесс-процесс. Даже по сравнению с защищенными объектами конструкции Java относительно слабы:

• Барьер защищенного объекта автоматически перевычисляется всякий раз, когда его значение может измениться; в Java нужно явно программировать циклы.

• Java предоставляет простую блокировку-пропускник для каждого объекта; в Ada заводится очередь для каждого входа. Таким образом, если несколько нитей Java ожидают входа в синхронный объект, вы не можете знать, какой из них будет запущен по notity, поэтому трудно программировать алгоритмы с гарантированно ограниченным временем ожидания.

18.7. Библиотеки Java

В языках программирования очевидна тенденция сокращения «размеров» языка за счет расширения функциональности библиотек. Например, write — это оператор в языке Pascal со специальным синтаксисом, тогда как в Ada нет никаких операторов ввода/вывода; вместо этого ввод/вывод поддерживается пакетами стандартной библиотеки.

Стандартные библиотеки Ada предоставляют средства для ввода/вывода, обработки символов и строк, для вычисления математических функций и для системных интерфейсов. Язык C++ также поддерживает контейнерные классы, такие как стеки и очереди. Точно так же Java содержит базисные библиотеки, названные java.lang, java.util и java.io, которые являются частью спецификации языка.

В дополнение к спецификации языка имеется спецификация для интерфейса прикладного программирования (Application Programming Interface — API), который поддерживают все реализации Java. API состоит из трех библиотек: Java.applet, Java.awt и java.net.

Java.applet поддерживает создание и выполнение аплетов и создание прикладных программ мультимедиа.

Абстрактный комплект инструментальных оконных средств (Abstract Window Toolkit — AWT) — это библиотека для создания графических интерфейсов пользователя (GUI): окна, диалоговые окна и растровая графика.

Библиотека для сетевой связи (java.net) обеспечивает необходимый интерфейс для размещения и пересылки данных по сети.

Подведем итог:

• Java — переносимый объектно-ориентированный язык с семантикой ссылки.

• Интерфейс прикладного программирования API представляет переносимые библиотеки для поддержки развития программного обеспечения в сетях.

• Защита данных и безопасность встроены в язык и модель.

• Многие важные концепции Java заложены в языково-независимую машину JVM.

8.8. Упражнения

1. Задано арифметическое выражение:

(a + b) * (c + d)

Java определяет, что оно должно вычисляться слева направо, в то время как C++ и Ada позволяют компилятору вычислять подвыражения в любом порядке. Почему в Java более строгая спецификация?

2. Сравните конструкцию final в Java с константами в Ada.

3. Каково соотношение между спецификатором friend в C++ и конструкцией пакета в Java.

4. C++ использует спецификатор protected (защищенный), чтобы разрешить видимость членов в порожденных классах. Как именно конструкция пакета влияет на понятие защищенности в Java?

5. Сравните интерфейс в Java с многократным наследованием в C++.

6. Проанализируйте различия между пространством имен (namespace) в C++ и пакетом в Java, особенно относительно правил, касающихся файлов и вложенности.

7. Конструкция исключения в Java совершенно аналогична конструкции исключения в C++. Одно важное различие состоит в том, что метод Java должен объявить все исключения, которые он может породить. Обоснуйте это проектное решение и обсудите его последствия.

8. Сравните мониторы Java с классической конструкцией монитора.

9. Сравните возможности обработки строк в Ada95, C++ и Java.

10. Сравните операции clone и eguals в Java с этими операциями в языке Eiffel.

Ссылки

Официальное описание языка дается в:
James Gosling, Bill Joy and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1997.
Sun Microsystems, Inc., где разработан язык Java, имеет Web-сайт, содержащий документацию и программное обеспечение: http://java.sun.com.

Приложение А

Где получить компиляторы

В течение многих лет студентам было сложно экспериментировать с языками программирования: компиляторы могут дорого стоить, и, возможно, не так просто убедить компьютерный центр установить и поддерживать программное обеспечение. Сегодня ситуация изменилась, и можно получить свободно распространяемые компиляторы для большинства, если не для всех, языков, которые мы обсудили.

Эти компиляторы предназначены для рабочих станций и даже персональных компьютеров, поэтому вы можете установить их дома или в своей лаборатории. Кроме того, они легко доступны через Internet.

Чтобы получать информацию относительно компиляторов для какого-либо языка, просмотрите файлы, называющиеся FAQ (Freguently Asked Questions — часто задаваемые вопросы). Их можно загрузить через анонимный ftp по адресу rtfm.mit.edu. В директории /pub/usenet находится (очень длинный) список поддиректорий; просмотрите comp.lang.x, где x — одно из ada, apl, c, c++, eiffel, icon, lisp, ml, prolog, Smalltalk и т.д. Переходите в одну и; этих поддиректорий и загружайте файлы, в которых есть символы FAQ. Эта файлы будут содержать списки доступных компиляторов языка, в частности те, которые можно загрузить, используя ftp. Хотя эти программы не представляют интереса как коммерческие пакеты, они вполне подходят для изучение и экспериментирования с языком.

В FAQ вы также найдете названия и адреса профессиональных ассоциаций, которые издают бюллетени и журналы, дающие современную информацию о языках.

C++

Свободно распространяемый компилятор для языка C++, называющийся gcc, был разработан в рамках проекта GNU компании Free Software Foundation. Более подробно см. FAQ в поддиректории gnu.g++.help на узле rtfm.mit.edu. Компилятор gcc был перенесен на большинство компьютеров включая персональные. Так как язык C++ все еще стандартизуется, gcc может отличаться от других компиляторов C++.

Ada 95

Нью-Йоркский университет разработал свободно распространяемый компилятор для языка Ada 95, названный gnat (GNU Ada Translator), gnat использует выходную часть gcc и перенесен почти на все компьютеры, которые поддерживают gcc. Современную информацию относительно ftp-сайтов для gnat см. в Ada FAQ; главный сайт находится в директории /pub/gnat в cs.nyu.edu. Там вы также найдете управляемую с помощью меню среду программирования для gnat, которая была разработана Университетом Джорджа Вашингтона.

AdaS

Pascal- S — это компилятор для подмножества языка Pascal, который вырабатывает *P-код,* являющийся машинным кодом для искусственной стековой машины. Включен также интерпретатор для P-кода. Его автор разработал версию Pascal-S, названную AdaS, которая компилирует небольшое подмножество языка Ada. Исходный код AdaS можно найти в файле adasnn.zip (где пп — номер версии) в каталоге /languages/ada/crsware/pcdp в общей библиотеке языка Ada (PAL) на хосте wuarchive.wustl.edu.

AdaS не годится для серьезного программирования, но это превосходный инструмент для изучения методов реализации конструкций языка программирования, в частности управления стеком при вызовах подпрограмм и возвратах из них.

Приложение Б

Библиографический список

Обзоры по языкам программирования можно найти в:
Ellis Horowitz (ed.). *Programming Languages: 4 Grand Tour.* Springer Verlag, 1983.
Jean E. Sammet. *Programming Languages: History and Fundamentals.* Prentice Hall, 1969.
Richard L. Wexelblat. *History of Programming Languages.* Academic Press, 1981.
Особенно интересна книга Векселблата (Wexelblat); это запись конференции, где разработчики первых языков программирования описывают истоки и цели своей работы.
Превосходное введение в теорию вычисления (логику, машины Тьюринга, формальные языки и верификацию программ) можно найти в:
Zohar Manna. *Mathematical Theory of Computation.* McGraw-Hill, 1974.
В учебниках, рассчитанных на подготовленных учащихся обсуждается формальная семантика языков программирования:
Michael Marcotty and Henry Legrand. *Programming Language Landscape: Syntax, Semantics and Implementation.* SRA, Chicago, 1986.
Bertrand Meyer. *Introduction to the Theory of Programming Languages.* Prentice Hall International, 1991.
По компиляции смотрите следующие работы:
Alfred Aho, Ravi Sethi and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.
Charles N. Fisher and Richard J. LeBlanc. *Grafting a Compiler.* Benjamin Cummings, 1988.
Хорошим введением в объектно-ориентированное проектирование и программирование является:
Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall International, 1988.
Обратите внимание, что описанная там версия языка Eiffel устарела; если вы хотите изучить язык, смотрите современное описание:
Bertrand Meyer. *Eiffel: the Language.* Prentice Hall, 1992.

Конкретные языки программирования

Мы даже не будем пытаться перечислить множество учебников по языкам C, Ada и C++!
Формальное описание языка Ada можно найти в справочном руководстве:
Ada 95Reference Manual. ANSI/ISO/IEC-8652:1995.
Справочное руководство очень формальное и требует тщательного изучения. Существует сопутствующий документ, называемый *Объяснением (Rationale),* в котором описана мотивация языковых конструкций и даны обширные примеры. Файлы, содержащие текст этих документов, можно бесплатно загрузить, как описано в Ada FAQ.
Стандарт языка C — ANS X3.159-1989; международный стандарт — ISO/IEC 9899:1990. В настоящее время (конец 1995 г.), язык C++ еще не стандартизирован; информацию о том, как получить последний предлагаемый вариант стандарта языка C++, см. в FAQ. Более доступно справочное руководство:

Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990 (reprinted 1994).

Следующая книга является «обоснованием» языка C++ и должна быть прочитана всеми серьезными студентами, изучающими этот язык:

Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, 1994.

Другие широко используемые объектно-ориентированные языки, которые стоит изучить, — Smalltalk и CLOS:

Adele Goldberg and David Robson. *Smalltalk-80, the Language and its Implementation.* Addison-Wesley, 1983.

Sonya E. Keene. *Object-Oriented Programming in Common Lisp: a Programmer's Guide.* Addison-Wesley, 1989.

В разделе 1.3 мы рекомендовали вам изучить один или несколько языков, основанных на конкретной структуре данных. Следующий список позволит вам начать это изучение:

Leonard Oilman and Alien J. Rose. APL: *An Interactive Approach. John* Wiley, 1984*.

Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language (2nd Ed.).* Prentice Hall, 1990.

J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL* Springer Verlag, 1986.

Patrick H.Winston and Berthold K.P.Horn. *LISP (3rd Ed.).* Addjson-Wesley, 1989.

Наконец, введение в языки и языковые понятия, которые были только кратце рассмотрены в этой книге, можно найти в:

M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice Hall International, 1990.

Ivan Bratko. *Prolog Programming for Artificial Intelligence (2nd Ed.).* Addison-Wesley, 1990.

Chris Reade. *Elements of Functional Programming.* Addison-Wesley, 1989. Leon SterUng and Ehud Shapiro. *The Art of Prolog.* MIT Press, 1986. Jeffrey D. Ullman. *Elements of ML Programming.* Prentice Hall, 1994.