Д. Хендрикс

Компилятор языка Си для микроЭВМ

The Small-C Handbook

James E. Hendrix

Office of Computing and Information Systems
The University of Mississippi

A Reston Computer Group Book
Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia

Д. Хендрикс

Компилятор языка Си для микроЭВМ

Перевод с английского А. А. Батнера
Под редакцией Б. А. Кузьмина

Москва
«Радио и связь»
1989

ББК 32.973
Х 38
УДК 681.3.068:519.682.7

Редакция автоматизации редакционно-издательских процессов

Хендрикс Д.
Х 38 Компилятор языка Си для микроЭВМ: Пер. с англ. - М.: Радио и связь, 1989. - 240 с.: ил.

ISBN 5-256-00161-2.

В книге американского автора приведены полный исходный текст компилятора языка Смолл-Си (подмножества языка Си), написанный на языке Смолл-Си, а также библиотека подпрограмм на языке ассемблера микропроцессоров Intel 8080 и 8086 для генератора кода компилятора.

Для программистов.

Х 2405000000-164 / 046(01)-89 150-88

ББК 32.973

ISBN 5-256-00161-2 (рус.)
ISBN 0-8359-7012-4 (англ.)

© 1984 by Reston Publishing Company, Inc.
© Перевод на русский язык, примечания переводчика.
Издательство "Радио и связь", 1989.

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Мы склонны любить тот язык программирования, который изучили первым, и с большой неохотой принимаем новые языки. По всей видимости, мы скорее согласимся терпеть какие-либо неудобства, чем идти по более правильному пути, особенно если это связано с изучением нового языка. Поэтому новый язык программирования, преодолевший эти тенденции и встретивший восторженный прием у опытных программистов, заслуживает особого внимания. Именно так происходит с языком Си. Его популярность быстро растет, и многие программисты, работающие на микроЭВМ, ищут дешевые пути приобщения к этому языку. Компилятор Смолл-Си, появившийся в 1980 г., удовлетворил эту потребность. В нем довольно экономично реализовано вполне приемлемое подмножество языка Си. Для применений, не требующих обработки действительных чисел, Смолл-Си имеет определенные преимущества перед Бейсиком и языком ассемблера. Программы, написанные на Смолл-Си, совместимы снизу вверх с компиляторами полной версии языка Си.

Тот, кто уже пользуется языком Смолл-Си или только еще подумывает им заняться, найдет здесь ценную информацию об этом языке и его компиляторе. Данный материал может быть интересен прежде всего трем категориям читателей:

1) программистам, использующим Смолл-Си и нуждающимся в руководстве по этому языку и компилятору;

2) программистам, использующим язык ассемблера, желающим повысить свою производительность и писать программы, которые можно переносить с одной ЭВМ на другую;

3) преподавателям и студентам по специальностям, связанным с вычислительной техникой. Небольшой размер компилятора и то, что сам он написан на Си, а не на языке ассемблера, делают Смолл-Си идеальной темой для лабораторных работ. Это реальный компилятор, достаточно простой для понимания и модификации студентами. Без особого труда он может быть превращен в кросскомпилятор или же полностью переведен на другой процессор. Можно расширить язык, внести усовершенствования и т. п. На этом маленьком компиляторе может базироваться любое число подобных работ.

Книга не является введением в программирование. Напротив, она дает описание языка и компилятора Смолл-Си для тех, кто уже программирует на других языках. По этой причине материал изложен довольно кратко и по существу.

В ч.1 рассматриваются основные понятия трансляции программ. ней представлен беглый обзор, затрагивающий центральный роцессор, машинный язык, язык ассемблера и использование ссемблеров, загрузчиков и компоновщиков программ. Дается нформация, достаточная для полного понимания кода на языке ссемблера, генерируемого компилятором. Этот материал рассчитан а использование архитектуры микропроцессора 8080, так как омпилятор, написанный первоначально для этого процессора, все це является наиболее популярной реализацией Смолл-Си. В гл.1 писан микропроцессор 8080, а в гл.3 дана его система команд. ем, кто близко не знаком с языком ассемблера, необходимо рочитать эти две главы.

В ч.2 рассматривается сам язык Смолл-Си. В главах этой асти в сжатой и точной форме последовательно описываются лементы языка, начиная с простых и кончая более сложными. оэтому эти главы удовлетворяют двум целям: они являются атким, но в то же время полным описанием языка и, кроме ого, служат справочным материалом.

В ч.3 описывается собственно компилятор. Здесь рассматрива-тся принципы организации ввода-вывода, стандартные функции, язов компилятора, генерация кода, приводятся некоторые ображения по увеличению эффективности программ и суждаются вопросы использования компилятора для генерации о новых версий.

В приложениях содержатся полные листинги исходных текстов омпилятора и библиотеки арифметических и логических одпрограмм, а также справочный материал для программиста, ботающего на языке Смолл-Си.

Я искренне признателен всем, кто поддерживал меня в этой боте. М. Ауверсон, которая задумалась о необходимости такой ниги и убедила в этом издателя. Р. Кейну, создавшему исход-ый компилятор Смолл-Си и давшему много полезных советов по овершенствованию данной версии. Э. Пейну, разработавшему ольшую часть библиотеки версии 2.1 компилятора для операци-ной системы CP/M. Н. Блоку за его помощь в разработке зличных функциональных элементов, начиная с версии 2.0, авным образом новых управляющих операторов языка. Э. Мак-ерсону и П. Уэсту за сообщения об обнаруженных ошибках и едостатках. Д. Уолману и Д. Бозуэллу, проверившим, соответст-нно, содержание и грамматику. Х. Фултону за помощь при боте с гранками. И в заключение - моей жене Гленде, столь рпеливо переносившей мое невнимание, пока я был занят этой ботой.

ВВЕДЕНИЕ

Язык программирования Си был разработан в начале 70-х годов Д. Ритчи из фирмы Bell Telephone Laboratories. Этот язык проектировался для того, чтобы получить непосредственный доступ к объектам, которыми оперируют процессоры ЭВМ: разрядам, байтам, словам и адресам. По этой причине, а также потому, что Си является блочно-структурированным языком, похожим на Алгол и Паскаль, он прекрасно подходит для системного программирования. Фактически на этом языке реализована операционная система UNIX.

Язык Си полезен также для других применений. Он очень удобен для обработки текстов, для технических приложений и моделирования. Конечно, другие языки имеют свои специфические особенности, которые делают их во многих случаях более удобными для определенных задач. Можно упомянуть операции с комплексными числами в Фортране, операции с матрицами в ПЛ/1, оператор сортировки и возможность выдачи отчетов в Коболе. Но тем не менее язык Си становится очень популярным, им широко пользуются программисты, и он им нравится.

Те, кто использует язык Си, приводят обычно следующие причины его популярности: 1) программы на языке Си легче перенести с одной ЭВМ на другую, чем большинство других программ; 2) язык Си обеспечивает богатый набор операций для вычисления выражений и дает возможность обходиться без языка ассемблера даже при работе с битами; 3) программы на языке Си компактны, но не настолько, чтобы быть непонятными; 4) язык Си включает в себя средства, позволяющие генерировать эффективный объектный код, и 5) язык Си удобен, его синтаксис достаточно прост.

Описание полного языка Си, работающего под управлением операционной системы UNIX, читатель может найти в книге B.W. Kernighan, D.M. Ritchie "The C Programming Language" (Prentice-Hall, 1978)[1].

В мае 1980 г. в журнале Dr. Dobb's Journal была опубликована статья "A Small C Compiler for the 8080s". В этой статье Р. Кейн предложил небольшой компилятор языка Си. Кроме малого объема наиболее интересной особенностью этого компилятора

[1] Перевод см. в книге: Керниган Б., Ритчи Д., Фьюэр А. Язык программирования Си. Задачи по языку программирования Си: Пер. с англ. - М. Финансы и статистика, 1985. - 279 с. - *Прим. перев.*

является то, что он был написан на том же языке, который компилирует. Таким образом, его можно было использовать для компиляции новых версий самого же компилятора. Несмотря на простой однопроходный алгоритм, этот компилятор генерировал код на языке ассемблера для микропроцессора 8080. Малым объемом определялись и некоторые ограничения компилятора. Он распознавал только символы и целые числа, а также одномерные массивы тех и других. Единственным оператором для управления циклом был оператор while. Отсутствовали булевы операторы, поэтому вместо них использовались поразрядные логические операторы :(ИЛИ) и &(И). Однако, несмотря на ограничения, это был весьма перспективный язык, столь же привлекательный, как и язык ассемблера.

Р. Кейн опубликовал листинг компилятора и передал его в общественный фонд программ. Как компилятор, так и язык стали известны под именем Смолл-Си. Этот компилятор вызвал большой интерес, и вскоре его можно было увидеть не только на миропроцессоре 8080, но и на других процессорах. Согласившись с необходимостью усовершенствования компилятора, Р. Кейн поддержал меня в разработке второй версии, и в декабре 1982 г. она появилась в журнале Dr. Dobb's Journal. Новый компилятор добавил к Смолл-Си следующие свойства: 1) оптимизацию кода, 2) инициализацию данных, 3) условную компиляцию, 4) класс переменных extern, 5) операторы for, do, while, switch и goto, 6) некоторые операции присваивания, 7) булевы операции, 8) операцию дополнения до единицы и другие средства. В этой книге описана модернизированная версия 2.1 компилятора и его языка.

В ч.1 приводится краткий обзор основных концепций трансляции программ. Если вы уже хорошо знакомы с использованием компиляторов, ассемблеров, загрузчиков и компоновщиков, то можете перейти сразу к ч.2. Однако если вы не знакомы с микропроцессором 8080, то вам следует сначала прочитать гл. 1 и 3, в которых описывается система команд микропроцессора 8080.

В ч.2 описан язык Смолл-Си, материал подается в порядке возрастания сложности. Каждый аспект языка обсуждается сам по себе и в связи с предыдущим материалом. Эта часть может служить как введением в изучение языка, так и справочником.

Часть 3 посвящена практическим моментам использования языка и компилятора.

В конце книги даны семь приложений. В приложении А приведен полный листинг компилятора, а в приложении Б - листинг библиотеки арифметических и логических подпрограмм. В приложении В перечислены возможные несоответствия с полной версией языка Си. В приложении Г содержатся список сообщений об ошибках, выдаваемых компилятором, и объяснения этих ошибок. В приложении Д приведена таблица символов кода ASCII. В приложении Е содержится справочный материал для программирования на языке ассемблера. Приложение Ж - это краткий справочник по языку Смолл-Си и библиотеке функций.

ЧАСТЬ 1

ОСНОВНЫЕ ПОНЯТИЯ ТРАНСЛЯЦИИ ПРОГРАММ

Термин *трансляция программы* означает весь процесс преобразования программы, написанной на исходном языке, в рабочее состояние. Этот процесс включает в себя два основных этапа: *генерацию* и *интерпретацию.*

Генерация - процесс перевода программы с одного языка на другой, более близкий к *машинному языку*, т.е. языку центрального процессора (ЦП) ЭВМ. Компиляторы, ассемблеры и загрузчики являются генерирующими трансляторами.

Интерпретация - заключительная стадия трансляции программы. В результате интерпретации программа, написанная на некотором языке, преобразуется в рабочее состояние. Этот этап называют также *выполнением программы.* Чтобы программа исполнила те функции, для которых она предназначена, ее необходимо вызвать на *выполнение.* Интерпретация может производиться другой программой (интерпретатором) или ЦП. Центральный процессор просматривает в памяти программу на машинном языке и выполняет те команды, которые находит. Этот этап всегда является последним в трансляции программы, так как даже если программа интерпретируется с помощью программного обеспечения, сам интерпретатор интерпретируется с помощью ЦП.

По-видимому, для понимания процесса трансляции программы лучше всего рассматривать этот процесс, начиная от ЦП. Так происходило исторически, и этот путь представляется наиболее естественным. Для экономии времени мы будем рассматривать современный ЦП Intel 8080 - тот самый ЦП, для которого была написана исходная версия компилятора Смолл-Си.

ГЛАВА 1

МИКРОПРОЦЕССОР 8080

На рис.1.1 представлена схема ЦП 8080 и памяти. Память можно рассматривать как простой массив 8-разрядных байтов. Каждый байт имеет некоторый уникальный адрес, который может быть выражен 16-разрядным целым числом без знака. Первый

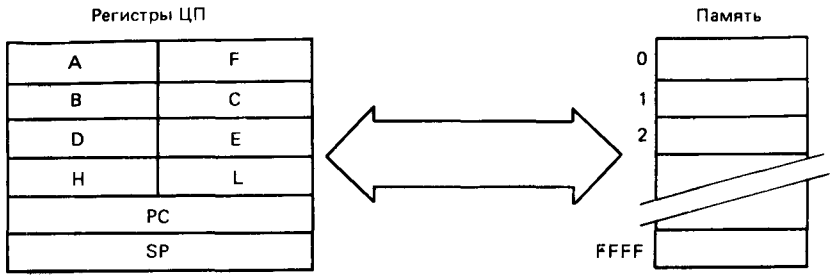

Рис.1.1. Архитектура микропроцессора 8080

айт имеет адрес 0, второй 1, третий 2, и т. д. Самый большой озможный адрес равен десятичному числу 65535 (шестнадцате-ичное число FFFF). Значения, хранящиеся в памяти, могут быть ибо данными, либо командами, управляющими работой ЦП.

Два последовательных байта составляют единое 16-разрядное исло и могут рассматриваться либо как данные, либо как адреса-ая часть команды. Такие длинные числа называют *словами*. Сло-а всегда хранятся в памяти таким образом, что первым идет ладший байт (этот байт имеет меньший адрес), а старший байт ледует за ним. Центральный процессор может читать значение айта или слова из памяти, перенося его в некоторый регистр ЦП (описанный ниже). При этом предыдущее содержимое регист-а теряется, а значение в памяти остается неизменным. Цент-альный процессор может также записывать значение в байт или слово памяти, перенося это значение из регистра. При этом одержимое регистра остается неизменным, а исходное значение в амяти заменяется новым. И при чтении, и при записи ЦП дол-ен переслать в память адрес желаемого байта или слова. Адре-ом слова является адрес его первого (младшего) байта.

Центральный процессор можно представить как некоторый на-ор *регистров* (мест для хранения данных), в которых временно аходятся значения байтов или слов. Обращение к регистрам роисходит быстрее, чем обращение к памяти, поэтому система оманд ЦП разрабатывается таким образом, чтобы придать ей ольшую гибкость при работе с регистрами. Регистры ЦП имеют ледующие имена: A, B, C, D, E, F, H, L, PC и SP. Размер егистров, имена которых состоят из одной буквы, составляет 8 азрядов, а регистры PC и SP имеют по 16 разрядов. В некото-ых командах регистры A, B, C, D, E, F, H и L рассматривают-я как 16-разрядные пары регистров AF, BC, DE и HL. В этом лучае первая буква соответствует старшему байту, а вторая - ладшему. Таким образом, когда 16-разрядное число находится в аре регистров HL, в регистре H содержатся старшие разряды, а регистре L - младшие.

Регистр F является специальным регистром, так как он не используется для хранения данных. Этот регистр представляет собой набор *флагов условий*; его разряды показывают, какие условия возникли в результате выполнения самой последней арифметической или логической команды. Каждый флаг имеет имя. Флаг нуля Z *установлен* (его значение равно единице), если результат выполнения последней арифметической или логической команды равен нулю; в противном случае он *очищен*, или *сброшен* (его значение равно нулю). Флаг знака S указывает на знак результата; он соответствует старшему разряду результата: единица для отрицательных значений и нуль - для положительных. Флаг переноса CY установлен, если был перенос из самого левого разряда или же заем в данный разряд; в противном случае этот флаг сброшен. Флаг четности P служит двум целям. Поразрядные логические команды устанавливают или очищают этот флаг в зависимости от четности результата. Если число единиц в результате *четное*, то флаг P устанавливается, если же число единиц *нечетное*, то флаг очищается. Если при выполнении 8-разрядных арифметических операций происходит переполнение, то флаг P устанавливается, иначе он очищается. Перечисленные выше разряды регистра флагов могут быть проверены с помощью команд *условного перехода*, *вызова подпрограммы* и *возврата*. Более подробно об этом будет сказано позже. Пару регистров AF называют также *словом состояния программы* (PSW), так как в регистре F содержится информация о состоянии.

Как было сказано выше, в памяти находятся как данные, так и команды. Команды указывают ЦП, какие операции и в каком порядке необходимо выполнять. Данные, на которые ссылаются команды, называются *операндами*. Длина команды может составлять 1, 2 или 3 байта памяти. В первом байте всегда содержится код, который указывает ЦП, какого рода операцию необходимо выполнить. Каждая конкретная команда, входящая в систему команд ЦП, идентифицируется таким *кодом операции*.

Команды, не содержащие ссылок на память, состоят только из кода операции. Эти команды пересылают данные между регистрами, работают с содержимым регистров, а также проверяют значения регистров.

Некоторые команды ссылаются на *непосредственный* операнд, т.е. операнд, входящий в состав самой команды. Если этот операнд занимает 1 байт памяти, то вся команда имеет длину 2 байта: код операции, за которым следует однобайтовый операнд. Если операнд занимает слово, то длина команды 3 байта: за кодом операции следует младший байт операнда, а затем старший байт.

Некоторые команды ссылаются на операнды в памяти по их адресам. При этом возможны два случая. В некоторых трехбайтовых командах после кода операции следует двухбайтовый адрес (сначала младший байт адреса, затем старший). Другие команды

ылаются на операнд, адрес которого содержится в регистре BC, E, HL или SP. Такие команды имеют длину 1 байт, так как адрес не является частью самой команды. Центральный процессор определяет длину команды по ее коду операции.

Регистр PC, или *счетчик команд*, указывает, какая команда должна быть выбрана следующей для исполнения. В нем находится адрес следующей команды. Обычно при выполнении команды содержимое регистра PC увеличивается в зависимости от длины команды на 1, 2 или 3. Следовательно, обычно порядок выполнения определяется порядком команд в памяти. Команды *перехода*, *вызова подпрограммы* и *возврата* изменяют этот порядок, помещая в регистр PC новый адрес.

Регистр SP является *указателем стека*. В нем содержится адрес слова, которое следует считать *вершиной стека*. Стек выполняет те же самые функции, что и очередь типа "последним пришел - первым обслужен". Таким образом, команды *записи в стек* и *чтения из стека* соответственно добавляют операнд в стек или удаляют его из стека. По мере записи операндов в стек вершина стека перемещается к началу памяти, в то же время основание стека остается фиксированным. Регистр SP всегда указывает на последний операнд, записанный в стек. Возможны запись в стек и чтение из стека только 16-разрядных значений.

Центральный процессор выполняет запись в стек за четыре шага: 1) уменьшает SP на единицу, 2) пересылает в память по адресу, заданному SP, старший байт регистра, 3) снова уменьшает SP и 4) пересылает в память по адресу, заданному SP, младший байт регистра. Чтение из стека происходит в обратном порядке: 1) байт с адресом, заданным SP, пересылается в младший байт регистра, 2) SP увеличивается на единицу, 3) байт с адресом, заданным SP, пересылается в старший байт регистра и 4) SP снова увеличивается.

Одно из важнейших применений стека - сохранение адреса возврата при вызове подпрограммы. Команды *вызова* подпрограммы являются особым видом перехода в подпрограмму. Эти команды записывают в стек содержимое регистра PC (т.е. адрес следующей команды), а затем выполняют переход в подпрограмму. Команды *возврата* используются внутри подпрограммы для передачи управления назад той команде, которая следует за командой вызова подпрограммы. Эти команды пересылают адрес из стека в регистр PC. Регистр PC определяет следующую команду, с которой продолжится работа, что эквивалентно переходу по адресу, находящемуся в вершине стека.

Стек используется программами, написанными на языке Смолл-Си, для размещения локальных переменных, передачи функциям аргументов и вызова функций.

Теперь, когда нам кое-что известно о том, как работает ЦП, можно обратиться к некоторым конкретным командам и посмотреть, как можно было бы их использовать. Предположим, что

нам необходима подпрограмма, которая будет просматривать некоторый блок памяти до тех пор, пока не обнаружит байт, имеющий шестнадцатеричное значение FF. На входе в подпрограмму регистр HL будет содержать адрес, с которого будет начат поиск. Для обращения к подпрограмме необходимо использовать команду вызова, в результате выполнения которой адрес возврата будет находиться в вершине стека. На выходе из подпрограммы в регистре HL будет содержаться адрес первого обнаруженного значения FF. Поиск, если необходимо, продолжится вплоть до максимально возможного адреса (шестнадцатеричный адрес FFFF), и, если значение FF не будет найдено, в регистре HL будет содержаться нуль.

Если бы эта подпрограмма была расположена в памяти, начиная с адреса 1000, то она могла бы выглядеть так, как показано в листинге 1.1. Чтобы можно было лучше понять работу программ, написанных на машинном языке, каждая команда помещена в отдельную строку вместе с адресом первого байта команды и комментарием, описывающим ее функцию.

```
Адрес    Значение  Комментарий
1000     06FF      переслать в B непосредственный операнд FF
1002     7E        переслать в A адрес байта, заданный в HL
1003     B8        сравнить A и B, если равны, установить Z
1004     C8        если Z установлен, вернуться из подпрограммы
1005     23        увеличить HL на единицу
1006     7C        переслать H в A
1007     B5        записать в A результат поразрядной операции
                   ИЛИ для регистров A и L, установить Z,
                   если результат равен нулю
1008     C8        если Z установлен, вернуться из подпрограммы
1009     C30210    перейти по адресу 1002 для следующей итерации
100C
```

Листинг 1.1. Пример подпрограммы на машинном языке

Теперь посмотрим, как работает эта подпрограмма. Команда 06, находящаяся по адресу 1000, пересылает в регистр B непосредственный операнд FF (второй байт команды). Это делается только один раз. Затем команда 7E пересылает в регистр A байт с адресом, содержащимся в HL. Команда B8 сравнивает значения в регистрах A и B, вычитая B из A и устанавливая в соответствии с результатом вычитания флаги. Сам результат вычитания не сохраняется. Если значения в регистрах A и B равны, то флаг Z будет установлен, иначе он будет сброшен. C8 является командой условного возврата, и возврат происходит только в том случае, если флаг Z установлен; в противном случае эта команда ничего не делает. Таким образом, если искомое значение найдено, то в

егистр PC заносится адрес из стека и следующей будет выполяться та команда, которая идет сразу за командой вызова подрограммы. В этом случае в регистре HL будет содержаться адрес айденного байта памяти. Если же значения не совпали, то слеующей будет выполнена команда 23, которая добавляет единицу содержимому пары регистров HL. Если последним сравнивался айт с адресом FFFF, то теперь в регистре HL будет содержаться дрес, равный нулю. Следующие три команды служат как раз для роверки этой ситуации: сначала команда 7C пересылает H в A; 5 выполняет поразрядную операцию ИЛИ для регистров A и L, омещая результат в A и устанавливая соответствующим образом лаги, и в заключение, если флаг установлен, команда C8 с адесом 1008 возвращает управление программе, обратившейся к анной подпрограмме. Если остались еще непроверенные ячейки амяти, то будет выполнена следующая трехбайтовая команда C3. Это команда безусловного перехода, содержащая адрес перехода, представленный в стандартной форме (первый байт - это младший байт адреса). Эта команда помещает число 1002 в регистр PC; таким образом, следующей будет выполняться команда, находящаяся по этому адресу. В результате цикл проверки повторяется, но теперь уже для следующего байта памяти. Рано или поздно одна из двух команд возврата C8 будет выполнена.

Из этого примера можно понять, что программирование на машинном языке было довольно скучным занятием. И дело не только в том, что трудно писать программы, пользуясь языком из одних чисел, но и в том, что по листингу программы на машинном языке без посторонней помощи трудно точно понять, что программа делает.

ГЛАВА 2

ОСНОВНЫЕ ПОНЯТИЯ ЯЗЫКА АССЕМБЛЕРА

Помощь программистам, использовавшим машинный язык, пришла от самой ЭВМ. Были разработаны программы для перевода с простых языков, состоящих из мнемонических обозначений и символов, на числовые машинные языки. Эти новые языки явились лишь первым шагом автоматизации программирования, так как каждое предложение такого языка соответствовало одной команде ЦП, что видно из примера подпрограммы, показанной на листинге 1.1. Такие трансляторы программ были названы *ассемблерами*, так как они ассемблируют (т.е. собирают) программу на машинном языке.

Язык ассемблера всегда отражает архитектуру соответствующего ЦП, так как он аналогичен машинному языку. Следовательно, языки ассемблера всегда являются машинно-зависимыми. У

каждого ЦП должен быть свой собственный язык ассемблера, а у некоторых ЦП (особенно это относится к микропроцессорам) бывает даже несколько языков ассемблера. Это означает, что разные реализации ассемблера для одного и того же процессора различаются между собой. Однако часто эти различия незначительны и не вызывают проблем несовместимости.

В этой главе рассматривается язык ассемблера микропроцессора 8080 вообще, а не какая-то его конкретная реализация. Будут описаны только те моменты, которые необходимы для понимания того, как язык ассемблера используется в компиляторе Смолл-Си. Если вы уже знакомы с языком ассемблера для какого-либо другого процессора, то можете пропустить эту главу и перейти сразу к гл. 3, в которой рассматривается система команд микропроцессора 8080.

Язык ассемблера предлагает некоторый набор средств, помогающих программисту в его работе. Два из них представляют наибольший интерес.

Во-первых, вы можете писать коды команд в виде буквенных обозначений. Систему команд легче запомнить, если использовать имена команд. Такие имена называются *мнемоническими кодами операций* или просто *мнемониками.*

Во-вторых, программист может вместо числовых адресов использовать символьные имена. Как показывает пример подпрограммы (см. листинг 1.1), если программист записывает на машинном языке команду перехода на определенную команду, то он должен знать адрес последней. Это означает, что прежде всего он должен решить, в каком месте памяти будет расположена данная программа, и использовать этот адрес в качестве адреса первой команды программы. Затем необходимо вычислить адрес каждой из последующих команд, добавляя длину предыдущей команды к ее адресу. Но и это не спасет от неприятностей, когда потребуется внести изменения в программу. Любое удаление или включение команд приводит к тому, что написанные ранее команды сдвигаются в памяти; в результате многие адреса команд становятся неверными, и будет необходимо найти все измененные адреса и вычислить их заново, причем на поиски одной ошибки можно потратить много часов. Все ассемблеры сами вместо программиста следят за адресами памяти. Если необходимо сослаться на какую-либо команду или операнд, то им присваивается *метка* (или имя), которая и используется повсюду вместо адреса. Ассемблер везде заменяет это имя на адрес данной команды или операнда.

Если приведенную выше подпрограмму переписать на языке ассемблера, то она станет более понятной. Результат показан на листинге 2.1.

Прежде всего заметим, что метки предшествуют только тем командам, на которые есть ссылки. На метку FIND ссылаются в других местах программы. Любая метка опознается по двоеточию,

```
Метка   Код операции   Операнды   Комментарий

FIND:   MVI            B,0FFH     ; переслать в B непосредственный операнд FF
LOOP:   MOV            A,M        ; переслать в A адрес байта, заданный в HL
        CMP            B          ; сравнить A и B, если равны, установить Z
        RZ                        ; если Z установлен, вернуться из подпрограммы
        INX            H          ; увеличить HL на единицу
        MOV            A,H        ; переслать H в A
        ORA            L          ; записать в A результат поразрядной
                                  ; операции ИЛИ для регистров A и L,
                                  ; установить Z, если результат равен нулю
        RZ                        ; если Z установлен, вернуться из подпрограммы
        JMP            LOOP       ; перейти по адресу 1002 для следующей итерации
```

Листинг **2.1**. Пример подпрограммы на языке ассемблера

которое следует непосредственно за меткой. Однако в некоторых ассемблерах двоеточия не требуются.

Далее заметим, что комментарии должны начинаться точкой с запятой. Большинство ассемблеров рассматривают точку с запятой как разделитель между предложением языка и комментариями. Они помещают комментарии в листинги программ, однако не обращают на них внимания. Заметим также, что команды делятся на два поля: код операции и список операндов. Этот список может содержать один или два операнда или же быть пустым (т.е. не содержать ни одного операнда). Коды операций специально выбраны так, чтобы обеспечить мнемоническую связь с названием функции, выполняемой соответствующей машинной командой. Операнды обычно обозначаются именами тех мест, где они хранятся (именами регистров или метками, указывающими на адреса памяти). Однако непосредственные операнды задаются константами. Например, команда

MVI B,0FFH

пересылает шестнадцатеричное значение FF в регистр B.

Нуль перед константой FF в приведенном выше примере обусловлен соглашением, что числовые константы всегда начинаются с десятичной цифры (0...9). Кроме того, если число записано не в десятичной системе счисления, то система счисления должна быть определена последним символом числа. В соответствии с этим буква H в конце числа задает шестнадцатеричную систему счисления. Операндом может быть комбинация констант и меток, составляющая выражение. Ассемблер сокращает каждое такое выражение до 8- или 16-разрядного двоичного числа и вставляет его в команду.

При пересылке операндов из одного места в другое (эти места указываются в поле операндов) перемещение всегда производится справа налево. Таким образом, команда

MOV A,H

пересылает содержимое регистра H в регистр A. Многие команды неявно используют аккумулятор. В этих случаях подразумевается использование регистра A, который не указывается в поле операндов. Примером может служить команда

CMP B

которая сравнивает B с A.

В приведенной выше подпрограмме команда перехода к следующей итерации вместо действительного числового адреса использует метку LOOP. В этом случае ассемблер заменит метку LOOP на шестнадцатеричное значение 1002 и поместит его в последние 2 байта команды перехода. Метка FIND дает подпрограмме имя, с помощью которого ее вызывают, точнее, имя адреса, используемого в команде вызова подпрограммы.

Многие ассемблеры позволяют задавать в строке только одну метку. В этом случае метка присваивается адресу следующей команды или операнда. Если в программе появляется несколько идущих подряд меток, то все они относятся к одному и тому же адресу, и их можно понимать как синонимы данного адреса. Компилятор Смолл-Си также предоставляет такую возможность, поэтому для совместной работы с ним должен использоваться и соответствующий ассемблер. Специальный символ $ является неявной ссылкой на адрес той команды, в которой этот символ появился. Таким образом, запись

JMP $-20

генерирует команду перехода по адресу, на 20 байтов меньшему адреса самой команды перехода.

Не все мнемоники генерируют машинные команды. Некоторые из них сообщают ассемблеру, что необходимо выполнить какие-либо специальные действия, например зарезервировать в памяти место для данных, расположить программу в определенном месте памяти, и другие, не относящиеся к основной задаче ассемблирования машинных команд. Такие мнемоники называют *директивами ассемблера* или *псевдооперациями*. Мы рассмотрим только шесть из них.

Часто желательно присвоить константам имена. Это не только дает возможность задавать их значения, но и облегчает внесение изменений в программу, так как для замены значения константы необходим всего лишь оператор, в котором имени присваивается значение. Директива

CONST: EQU 33H

отождествляет имя CONST с шестнадцатеричным числом 33. Везде, где появится имя CONST, ассемблер подставит шестнадцатеричное значение 33. Директива

ORG 100H

дает ассемблеру начальный адрес для вычисления последующих адресов. Она означает, что таким начальным адресом является шестнадцатеричное число 100. Обычно такая директива записывается в начале программы для задания расположения программы в памяти. Директива

BUF: DS 80

сообщает ассемблеру, что необходимо зарезервировать в памяти 80 (десятичное число) байтов и присвоить метку BUF первому байту. Мнемоника DS означает, что требуется определить место в памяти. Директива

BYTE: DB 0

приводит к тому, что ассемблер определяет байт в памяти, записывает в него значение нуль и называет его BYTE. В поле операндов может быть задана строка значений. Директива

DB 1,2,3,'A'

резервирует в памяти 4 байта, содержащих соответственно значения 1, 2, 3 и код ASCII буквы A. Директива

DB 'HARRY',0

определяет 5 байтов, содержащих строку HARRY в коде ASCII и нуль. Подобным же образом директива

DW -13,0,5

определяет три слова, содержащих значения -13, 0 и 5.

И наконец, директива END сообщает ассемблеру, что он достиг конца программы.

Проведенный обзор языка ассемблера был довольно беглым. Этого, однако, достаточно для понимания выходной информации, генерируемой компилятором Смолл-Си. Для получения более подробной информации я рекомендую книгу A.R. Miller "8080/Z80 Assembly Language" (John Wiley & Sons, 1981).

В следующей главе даются команды микропроцессора 8080 в формате языка ассемблера и описывается работа этих команд.

ГЛАВА 3

СИСТЕМА КОМАНД МИКРОПРОЦЕССОРА 8080

В данной главе представлены все команды микропроцессора 8080, в том числе и те, которыми компилятор Смолл-Си не пользуется. Каждая команда приводится в таблице вместе с ее форматом, функциональным описанием и воздействием на флаги. Однотипные команды собраны в одной таблице (табл.3.3-3.13). В табл.3.1 приведены обозначения, используемые в других таблицах.

Таблица 3.1. Символы, применяемые при описании команд

Символ	Значение
=	Замена
<=>	Обмен операндов
_	Логическое объединение элементов
<AND>	Поразрядная операция И
<OR>	Поразрядная операция ИЛИ
<XOR>	Поразрядная операция исключающее ИЛИ
<NOT>	Дополнение до единицы
[x]	Однобайтовый операнд в порте ввода-вывода x
(x)	Однобайтовый операнд в ячейке памяти x
(x,y)	Двухбайтовый операнд в ячейках памяти x и y (x относится к старшему байту)
M	Операнд в ячейке памяти, адрес которой задан в регистре HL
*	Флаг изменяется в соответствии с результатом
V	При переполнении условие PE, иначе PO
IE	Триггер разрешения прерывания
CY	Флаг переноса
S	Знаковый разряд
n	Однобайтовое целое без знака (в диапазоне десятичных чисел 0...255)
nn	Двухбайтовое целое без знака (в диапазоне десятичных чисел 0...65535)
d	Однобайтовое целое со знаком (в диапазоне десятичных чисел -128...0...+127)
p	0H, 8H, 10H, 18H, 20H, 28H, 30H или 38H
cc	C, NC, Z, NZ, PE, PO, M или P (мнемонические значения флагов)
r	A, B, C, D, E, H или L
r'	A, B, C, D, E, H или L
rm	A, B, C, D, E, H, L или M
rr	B, D, H или SP (пары регистров)
ss	B, D, H или PSW (пары регистров)

Таблица 3.2. Длины команд

Длина команды, байт	Тип ссылки на операнд
1	Нет операндов
	Операнд в каком-либо регистре
	Адрес операнда в каком-либо регистре
2	8-разрядный непосредственный операнд
3	16-разрядный непосредственный операнд
	16-разрядный непосредственный адрес

Если в какой-либо таблице содержатся команды, изменяющие флаги (см., например, табл.3.6), то для каждого флага выделяется отдельная колонка. Заголовки таких колонок состоят из двух строк, в которых даны реальные мнемонические коды, используемые в командах проверки флага. Код в первой строке соответствует *установленному* флагу, во второй - *очищенному*. В форматы команд, проверяющих эти флаги (табл.3.11 и 3.12), входят символы cc. При записи такой команды вместо символов cc должен быть подставлен один из кодов, приведенных в заголовке.

Некоторые команды ссылаются на определенные разряды байта. При этом считается, что разряды нумеруются в порядке 7-6-5-4-3-2-1-0. Таким образом, ссылка на разряд 7 относится к старшему, т.е. знаковому, разряду.

Так как числовые значения кодов операций обычно не важны для программистов, пишущих программы на языке ассемблера, они здесь не приводятся. Не даются также и длины команд, хотя иногда их надо знать. Так как длина команды зависит, как правило, от способа ссылки на операнды, многие из мнемоник, рассматриваемых ниже, генерируют команды различной длины. Поэтому вместо того, чтобы связывать длину команды с мнемоникой, в табл.3.2 приведены некоторые приблизительные правила определения длины команды.

Вероятно, вам захочется еще раз вернуться к этой главе, когда вы будете изучать команды, генерируемые компилятором (гл.18), и рассматривать библиотеку арифметических и логических подпрограмм (приложение Б).

Таблица 3.3. 8-разрядные команды загрузки

Формат команды	Функциональное описание	Формат команды	Функциональное описание
LDA nn	A = (nn)	STA nn	(nn) = A
LDAX B	A = (BC)	STAX B	(BC) = A
LDAX D	A = (DE)	STAX D	(DE) = A
MVI r,n	r = n	MVI M,n	(HL) = n
MOV r,r'	r = r'	MOV M,r	(HL) = r
MOV r,M	r = (HL)		

Команда LDA nn загружает в A байт из ячейки памяти nn. Таким образом, команда LDA 123 пересылает в A байт из ячейки памяти с десятичным адресом 123, а команда LDA MYBYTE загружает в A байт, адресуемый меткой MYBYTE.

Команда LDAX B загружает в A байт, адрес которого содержится в паре регистров BC.

Команда LDAX D загружает в A байт, адрес которого содержится в паре регистров DE.

Команда MVI r,n пересылает непосредственное значение n (1 байт) в регистр r. Таким образом, команда MVI C,0 пересылает 0 в C.

Команда MOV r,r' пересылает содержимое одного 8-разрядного регистра в другой 8-разрядный регистр. Таким образом, команда MOV A,B передает содержимое регистра B в A, в результате чего в обоих регистрах будет содержаться одно и то же.

Команда MOV r,M пересылает 1 байт памяти, адресуемый по регистрам HL, в регистр r. Таким образом, команда MOV E,M пересылает байт, на который указывают регистры HL, в E.

Команда STA nn записывает байт из регистра A в ячейку памяти nn. Таким образом, команда STA 010A3H пересылает байт из регистра A в ячейку с шестнадцатеричным адресом 10A3.

Команда STAX B записывает содержимое регистра A в ячейку памяти по адресу, содержащемуся в паре регистров BC.

Команда STAX D записывает содержимое регистра A в ячейку памяти, на которую указывает пара регистров DE.

Команда MVI M,n пересылает непосредственное значение n (1 байт) в ячейку памяти, на которую указывают регистры HL. Таким образом, команда MVI M,32 пересылает десятичное значение 32 в ячейку памяти, адресуемую по регистрам HL.

Команда MOV M,r пересылает байт из регистра r в ячейку памяти, на которую указывает пара регистров HL. Таким образом, команда MOV M,L пересылает в ячейку памяти младшие 8 разрядов ее же собственного адреса.

Таблица 3.4. 16-разрядные команды загрузки

Формат команды	Функциональное описание
LXI rr,nn	rr = nn
LHLD nn	HL = (nn+1,nn)
SHLD nn	(nn+1,nn) = HL
SPHL	SP = HL
PUSH ss	(SP-1,SP-2) = ss SP = SP - 2
POP ss	ss = (SP+1,SP) SP = SP + 2

Команда LXI rr,nn загружает непосредственный операнд двойной длины (2 байта) nn в пару регистров rr. Таким образом,

команда LXI H,ARRAY+1 загружает в регистры HL адрес, являющийся результатом добавления единицы к адресу ARRAY: В регистрах HL будет содержаться адрес байта, следующего за ARRAY.

Команда LHLD nn загружает в пару регистров HL 16-разрядный операнд из ячейки памяти с адресом nn. В регистр L попадает байт из ячейки nn, а в регистр H - байт из ячейки nn+1. Таким образом, команда LHLD COUNTER пересылает в регистры HL 16-разрядное значение из ячейки памяти с адресом COUNTER.

Команда SHLD nn записывает содержимое регистров HL в ячейки памяти nn и nn+1, при этом содержимое регистра L попадает в ячейку nn, а содержимое регистра H - в ячейку nn+1. Таким образом, команда SHLD COUNTER записывает 16-разрядное значение из регистров HL в ячейку памяти с адресом COUNTER.

Команда SPHL пересылает содержимое регистров HL в регистр SP. Эта команда не имеет явных операндов, так как использование регистров SP и HL определяется кодом операции. Это прямой путь для установки указателя стека на заданную ячейку памяти.

Команда PUSH ss записывает операнд двойной длины из пары регистров ss в стек. Эта команда делает следующее: 1) уменьшает содержимое SP на единицу, 2) пересылает старший байт в ячейку памяти, адресуемую SP, 3) еще раз уменьшает содержимое SP и 4) пересылает младший байт в ячейку памяти, адресуемую SP. После выполнения команды в SP остается новое значение, а исходное содержимое регистра-источника не изменяется. Таким образом, команда PUSH D записывает в стек содержимое регистров DE.

Команда POP ss пересылает операнд двойной длины из вершины стека в пару регистров ss. Эта команда действует следующим образом: 1) байт, адресуемый SP, пересылается в младший регистр, 2) содержимое SP увеличивается на единицу, 3) байт, адресуемый SP, пересылается в старший регистр и 4) содержимое SP снова увеличивается. После выполнения команды SP приобретает новое значение. Таким образом, команда POP B записывает в регистры BC 16-разрядный операнд из вершины стека.

Таблица 3.5. **Команды обмена**

Формат команды	Функциональное описание
XCHG	DE<=>HL
XTHL	HL<=>(SP+1,SP)

Команда XCHG обменивает содержимое регистров DE и HL.

Команда XTHL обменивает содержимое регистров HL на 16-разрядный операнд, находящийся в вершине стека. Указатель стека не изменяется.

Таблица 3.6. **Команды 8-разрядной арифметики**

Формат команды		Функциональное описание	Условия C NC	Z NZ	PE PO	M P
ADI	n	A = A + n	*	*	V	*
ADD	rm	A = A + rm	*	*	V	*
ACI	n	A = A + n + CY	*	*	V	*
ADC	rm	A = A + rm + CY	*	*	V	*
SUI	n	A = A - n	*	*	V	*
SUB	rm	A = A - rm	*	*	V	*
SBI	n	A = A - n - CY	*	*	V	*
SBB	rm	A = A - rm - CY	*	*	V	*
DAA		Коррекция A	*	*	*	*
INR	rm	rm = rm + 1	*	V	*	
DCR	rm	rm = rm - 1	*	V	*	

Команда ADI n прибавляет к содержимому регистра A 8-разрядный непосредственный операнд n. Все флаги условий, включая флаг переноса, устанавливаются в соответствии с результатом. Флаг переноса, однако, не участвует в сложении. Таким образом, команда ADI 5 прибавляет 5 к содержимому регистра A, помещая сумму в регистр A.

Команда ADD rm прибавляет к содержимому регистра A 8-разрядный операнд, содержащийся в регистре rm. Все флаги условий, включая флаг переноса, устанавливаются в соответствии с результатом. Флаг переноса, однако, не участвует в сложении. Таким образом, команда ADD C прибавляет содержимое регистра C к содержимому регистра A, помещая сумму в регистр A, а команда ADD M прибавляет к содержимому регистра A содержимое ячейки памяти, на которую указывают регистры HL, и помещает сумму в регистр A.

Команда ACI n прибавляет к содержимому регистра A 8-разрядный непосредственный операнд и значение флага переноса, помещая сумму в регистр A. Все флаги условий устанавливаются в соответствии с результатом. Таким образом, команда ACI 3 прибавляет к содержимому регистра A число 3 и значение флага CY.

Команда ADC rm прибавляет к содержимому регистра A содержимое регистра rm и значение флага переноса, помещая сумму в регистр A. Все флаги условий устанавливаются в со-

ответствии с результатом. Таким образом, команда ADC B прибавляет к содержимому регистра A содержимое регистра B и CY, а команда ADC M прибавляет к содержимому регистра A байт, находящийся в ячейке памяти, на которую указывают регистры HL, и CY.

Команда SUI n вычитает из содержимого регистра A 8-разрядный непосредственный операнд n, помещая разность в регистр A. Все флаги условий, включая флаг переноса, устанавливаются в соответствии с результатом. Флаг переноса, однако, не участвует в вычитании. Таким образом, команда SUI 23 вычитает десятичное число 23 из содержимого регистра A.

Команда SUB rm вычитает из содержимого регистра A содержимое регистра rm, помещая разность в регистр A. Все флаги условий, включая флаг переноса, устанавливаются в соответствии с результатом. Флаг переноса, однако, не участвует в вычитании. Таким образом, команда SUB E вычитает содержимое регистра E из содержимого регистра A, а команда SUB M вычитает из содержимого регистра A содержимое ячейки памяти, на которую указывают регистры HL.

Команда SBI n вычитает из содержимого регистра A 8-разрядный непосредственный операнд и значение флага переноса. Все флаги условий устанавливаются в соответствии с результатом. Таким образом, команда SBI 1AH вычитает из содержимого регистра A шестнадцатеричное число 1A и CY.

Команда SBB rm вычитает из содержимого регистра A содержимое регистра rm и значение флага переноса. Все флаги условий устанавливаются в соответствии с результатом. Таким образом, команда SBB H вычитает из содержимого регистра A содержимое регистра H и CY.

Команда DAA выполняет десятичную коррекцию содержимого регистра A. Эта команда применяется после команд 8-разрядного сложения и вычитания для преобразования результата в два двоично-десятичных числа (полубайта) по четыре разряда в каждом. Выполнение этой команды связано с использованием не упоминавшегося ранее флага - флага вспомогательного переноса AC. При 8-разрядном сложении и вычитании этот флаг действует аналогично флагу CY, с той лишь разницей, что в этот флаг попадает не перенос из разряда 7 (старшего разряда байта) при сложении (или заем в разряде 7 при вычитании), а перенос (или заем), связанный с разрядом 3 (старшим разрядом младшего полубайта). Хотя флаг AC изменяют многие команды, команда DAA является единственной командой, которая этот флаг использует. Если младший полубайт регистра A больше девяти или же установлен флаг AC, то команда DAA прибавляет к младшему полубайту 6, после чего эта команда то же самое проделывает и со старшим полубайтом, если он больше девяти или же установлен флаг CY.

Команда INR rm увеличивает на единицу содержимое регистра rm. Изменяются все флаги, за исключением флага CY. Таким образом, команда INR D прибавляет единицу к содержимому регистра D, а команда INR M прибавляет единицу к содержимому ячейки памяти, на которую указывают регистры HL.

Команда DCR rm уменьшает на единицу содержимое регистра rm. Изменяются все флаги, за исключением флага CY. Таким образом, команда DCR E вычитает единицу из содержимого регистра E, а команда DCR M вычитает единицу из содержимого ячейки памяти, на которую указывают регистры HL.

Таблица 3... **Команды 16-разрядной арифметики**

Формат команды	Функциональное описание	Условия C NC
DAD rr	HL = HL + rr	*
INX rr	rr = rr + 1	
DCX rr	rr = rr - 1	

Команда DAD rr выполняет сложение операндов двойной длины. Эта команда прибавляет к содержимому регистров HL содержимое пары регистров rr, помещая сумму в регистры HL. Единственный флаг, на который влияет это команда, - CY. Таким образом, команда DAD D заменяет содержимое регистров HL суммой содержимого регистров HL и DE, а команда DAD H удваивает значение содержимого регистров HL.

Команда INX rr увеличивает на единицу содержимое пары регистров rr. Ни один из флагов не изменяется. Таким образом, команда INX H прибавляет единицу к содержимому регистров HL.

Команда DCX rr уменьшает на единицу содержимое пары регистров rr. Ни один из флагов не изменяется. Таким образом, команда DCX B вычитает единицу из содержимого регистров BC.

Таблица 3.8. **Логические команды**

Формат команды	Функциональное описание	Условия C Z PE M NC NZ PO P
ANI n	A = A<AND>n	NC * * *
ANA rm	A = A<AND>nm	NC * * *
ORI n	A = A<OR>n	NC * * *

Табл.3.8 (окончание)

Формат команды	Функциональное описание	Условия C Z PE M NC NZ PO P
ORA rm	A = A<OR>nm	NC * * *
XRI n	A = A<XOR>n	NC * * *
XRA rm	A = A<XOR>nm	NC * * *
CPI n	A - n	* * V *
CMP rm	A - rm	* * V *
CMA	A = <NOT>A	
CMC	CY = <NOT>CY	*
STC	CY = 1	C

Команда ANI n выполняет поразрядную логическую операцию И для регистра A и 8-разрядного непосредственного операнда n, помещая результат в регистр A. Эта команда очищает флаг CY, устанавливая тем самым условие NC. Все остальные флаги условий устанавливаются в соответствии с результатом. Термин *поразрядная* означает, что операция обрабатывает соответствующие разряды операндов: первые разряды обоих операндов определяют первый разряд результата, вторые разряды операндов определяют второй разряд результата, и т.д. Таким образом, команда ANI 7FH очищает старший разряд регистра A, не изменяя всех остальных разрядов.

Команда ANA rm выполняет поразрядную операцию И для регистров A и rm, помещая результат в регистр A. Эта команда очищает флаг CY, устанавливая тем самым условие NC. Все остальные флаги условий устанавливаются в соответствии с результатом. Таким образом, команда ANA B помещает в регистр A результат поразрядной операции И для регистров A и B, а команда ANA M помещает в регистр A результат поразрядной операции И для регистра A и байта памяти, на который указывают регистры HL.

Команда ORI n выполняет поразрядную операцию ИЛИ для регистра A и 8-разрядного непосредственного операнда n, помещая результат в регистр A. Эта команда очищает флаг CY, устанавливая тем самым условие NC. Все остальные флаги условий устанавливаются в соответствии с результатом. Таким образом, команда ORI 80H устанавливает старший разряд регистра A, не изменяя всех остальных разрядов.

Команда ORA rm выполняет поразрядную операцию ИЛИ для регистров A и rm, помещая результат в регистр A. Эта команда очищает флаг CY, устанавливая тем самым условие NC. Все остальные флаги условий устанавливаются в соответствии с результатом. Таким образом, команда ORA D помещает в регистр A результат поразрядной операции ИЛИ для регистров A и D, а команда ORA M помещает в регистр A результат поразрядной операции ИЛИ для регистра A и байта памяти, на который указывают регистры HL.

Команда XRI n выполняет поразрядную операцию исключающее ИЛИ для регистра A и 8-разрядного непосредственного операнда n, помещая результат в регистр A. Эта команда очищает флаг CY, устанавливая тем самым условие NC. Все остальные флаги условий устанавливаются в соответствии с результатом. Таким образом, команда XRI OFH инвертирует младший полубайт регистра A.

Команда XRA rm выполняет поразрядную операцию исключающее ИЛИ для регистров A и rm, помещая результат в регистр A. Эта команда очищает флаг CY, устанавливая тем самым условие NC. Все остальные флаги условий устанавливаются в соответствии с результатом. Таким образом, команда XRA A выполняет операцию исключающее ИЛИ, когда в качестве обоих операндов задан регистр A. Такая команда применяется обычно для очистки аккумулятора, так как она занимает всего лишь 1 байт и не обращается за операндами к памяти.

Команда CPI n сравнивает 8-разрядный непосредственный операнд n с содержимым регистра A, вычитая этот операнд из содержимого регистра A и устанавливая соответствующим образом флаги. Содержимое аккумулятора при этом не изменяется. Таким образом, команда CPI 0 сравнивает 0 с содержимым регистра A.

Команда CMP rm сранивает содержимое регистров rm и A, вычитая одно из другого и устанавливая соответствующим образом флаги. Содержимое аккумулятора при этом не изменяется. Таким образом, команда CMP B сравнивает содержимое регистров B и A, а команда CMP M сравнивает байт памяти, на который указывают регистры HL, с содержимым регистра A.

Команда CMA выполняет инвертирование содержимого регистра A. Каждый разряд, установленный в единицу, сбрасывается в нуль, и наоборот. Команда не влияет ни на один флаг.

Команда CMC инвертирует флаг переноса. Остальные флаги не изменяются.

Команда STC устанавливает флаг переноса. Остальные флаги не изменяются.

Таблица 3.9. Команды управления микропроцессором

Формат команды	Функциональное описание
NOP	Нет операции
HLT	Останов микропроцессора
DI	IE = 0
EI	IE = 1

Команда NOP - однобайтовая команда *нет операции.* При ее выполнении ничего не происходит, она всего-навсего занимает место в программе. Эта команда применяется для того, чтобы ставить "заплаты" в выполняемой программе и оставлять место для будущих "заплат".

Команда HLT останавливает микропроцессор.

Команда DI запрещает прерывания, очищая триггер разрешения прерываний.

Команда EI разрешает прерывания, устанавливая триггер разрешения прерываний.

Таблица 3.10. **Команды циклического сдвига**

Формат команды	Функциональное описание	Условия C NC
RLC	Циклический сдвиг A влево	*
RAL	Циклический сдвиг CY_A влево	*
RRC	Циклический сдвиг A вправо	*
RAR	Циклический сдвиг A_CY вправо	*

Команда RLC циклически сдвигает содержимое регистра A на одну позицию влево. Эта команда пересылает каждый разряд на одну позицию влево и помещает старший разряд в младшую позицию. Кроме того, старший разряд попадает во флаг переноса CY. Никакие другие флаги не изменяются.

Команда RAL циклическии сдвигает содержимое регистра A на одну позицию влево, используя флаг CY. Флаг CY оказывается в младшем разряде, старший разряд попадает во флаг переноса CY, а содержимое регистра A сдвигается влево. Никакие другие флаги не изменяются.

Команда RRC циклическии сдвигает содержимое регистра A на одну позицию вправо. Эта команда пересылает каждый разряд на одну позицию вправо и помещает младший разряд в старшую позицию. Кроме того, младший разряд попадает во флаг переноса CY. Никакие другие флаги не изменяются.

Команда RAR циклически сдвигает содержимое регистра A на одну позицию вправо, используя флаг CY. Флаг CY попадает в старший разряд, младший разряд попадает во флаг переноса CY, а содержимое регистра A сдвигается вправо. Никакие другие флаги не изменяются.

Таблица 3.11. **Команды перехода**

Формат команды	Функциональное описание
JMP nn	PC = nn
Jcc nn	PC = nn, если значение cc "истина"
PCHL	PC = HL

Команда JMP nn передает управление следующей команде по адресу nn. Это называется безусловным переходом, так как результат не зависит от значений флагов условий. Адрес назначения nn является непосредственным операндом, передаваемым в счетчик команд PC, в результате чего выбирается следующая выполняемая команда.

Команда Jcc nn передает управление по адресу nn только в том случае, если значение cc "истина". Вместо cc в этой команде может быть использована любая из мнемоник условий (C, NC, Z, NZ, P, M, PE, PO). Таким образом, команда JNZ THERE передает управление по адресу THERE только в том случае, если флаг Z не установлен; иначе управление передается на следующую команду.

Команда PCHL пересылает содержимое регистров HL в регистр PC. В результате происходит безусловный переход по адресу, содержащемуся в регистрах HL.

Таблица 3.12. **Команды вызова и возврата**

Формат команды	Функциональное описание
CALL nn	(SP-1, SP-2) = PC SP = SP - 2 PC = nn
Ccc nn	вызов, если значение cc "истина"
RET	PC = (SP+1,SP) SP = SP + 2
Rcc	возврат, если значение cc "истина"
RST p	(SP-1, SP-2) = PC SP = SP - 2 PC = p

Команда CALL nn вызывает подпрограмму по адресу nn. Эта команда записывает в стек адрес следующей команды (адрес возврата), а затем выполняет переход на nn. Таким образом, для вызова подпрограммы, приведенной в листинге 2.1, можно воспользоваться командой CALL FIND.

Команда Ccc nn является условной командой вызова, выполняющей вызов подпрограммы только в том случае, когда значение cc "истина". Вместо cc в этой команде может быть любая из мнемоник условий (C, NC, Z, NZ, P, M, PE, PO). Таким образом, команда CZ FIND вызывает подпрограмму FIND только в том случае, если флаг Z установлен; иначе управление сразу передается на следующую команду.

Команда RET возвращает управление из подпрограммы в то место, откуда подпрограмма была вызвана. Предполагается, что в вершине стека содержится правильный адрес возврата. Команда RET пересылает содержимое вершины стека в регистр PC, обеспечивая затем переход по адресу, заданному в регистре PC.

Команда Rcc является командой условного возврата, выполняемой только в том случае, когда значение cc "истина". Вместо cc в этой команде может быть любая из мнемоник условий (C, NC, Z, NZ, P, M, PE, PO). Таким образом, команда RNC возвращает управление только тогда, когда флаг CY очищен.

Команда RST p является специальной командой вызова, называемой *рестарт*. В трех разрядах кода операции этой команды содержится адрес перехода. Эти три разряда попадают в разряды 3, 4 и 5 регистра PC, поэтому команда RST p может выполнять вызов только по адресам 0H, 8H, 18H, 20H, 28H, 30H и 38H.

Таблица 3.13. **Команды ввода-вывода**

Формат команды	Функциональное описание
IN n	A = [n]
OUT n	[n] = A

Микропроцессор 8080 позволяет обращаться к 256 портам ввода-вывода, т.е. путям, через которые могут быть переданы данные между аккумулятором и внешними устройствами. Независимо от числа и назначения портов каждому из них присваивается уникальный адрес в диапазоне 0...255. Для связи с внешними устройствами предназначены две команды: IN n, которая читает байт из порта ввода-вывода n, и OUT n, которая записывает байт в порт ввода-вывода n.

В разных ЭВМ номера портов устройств отличаются. Чтобы понять результаты работы команд IN n и OUT n вашей конкретной ЭВМ, вам необходимо обратиться к техническому описанию ЭВМ. Использование этих команд делает программу зависящей не только от типа микропроцессора, но и от типа ЭВМ. Первое правило написания программ, которое можно переносить с одной ЭВМ на другую, - это избегать применения таких команд. Обычно их используют только в драйверах устройств, являющихся частью операционной системы, а пользовательские команды для выполнения ввода-вывода обращаются уже к операционной системе.

ГЛАВА 4

СРЕДСТВА ДЛЯ ТРАНСЛЯЦИИ ПРОГРАММ

Операторы, из которых состоит программа, обычно называют *кодом* программы. Операторы исходного языка, на котором написана программа, называют *исходным кодом*. Ассемблеры читают исходный код и генерируют *объектный код*. Компиляторы также читают исходный код и генерируют объектный код. Однако некоторые компиляторы генерируют *промежуточный код*, который затем должен пройти еще один этап трансляции программы. Например, Смолл-Си генерирует промежуточный код на языке ассемблера, который необходимо потом ассемблировать в объектный код. Промежуточным кодом называют также программу во внутреннем представлении, создаваемом во время первого прохода многопроходным компилятором или транслятором. Хотя компилятор Смолл-Си и является однопроходным, можно считать, что он выполняет как бы первый проход многопроходного компилятора, состоящего из компилятора Смолл-Си и используемого совместно с ним ассемблера.

Абсолютные ассемблеры генерируют программу, готовую к исполнению в определенных адресах памяти. Абсолютный объектный код подпрограммы FIND (см. листинги 1.1 и 2.1), выраженный в шестнадцатеричном формате, выглядит следующим образом:

06FF7EB8C8237CB5C8C30210

Этот код, загруженный в память начиная с ячейки с шестнадцатеричным адресом 1000, будет правильно исполняться. Такое числовое представление программы называется *отображением в памяти* (или *двоичным отображением*, так как оно отображает программу в том виде, в котором она существует во время исполнения).

Не всегда удобно, чтобы ассемблер генерировал программу прямо в память, в которой она будет находиться во время выполнения. Как правило, бывает необходимо сохранять объектные программы в файлах, чтобы не ассемблировать их при каждом выполнении. Таким образом, необходим некоторый *загрузчик* для пересылки программ из объектных файлов в память перед их выполнением.

Наиболее наглядным форматом объектного файла является простое отображение программы в памяти в момент выполнения, так как в этом случае каждый байт файла представляет собой соответствующий байт памяти. Такое представление называется *двоичным форматом*, так как оно является точной копией двоичной программы в памяти. Для такого объектного файла нужен самый примитивный загрузчик. Он просто загружает объектный файл в память по тем адресам, где будет исполняться программа.

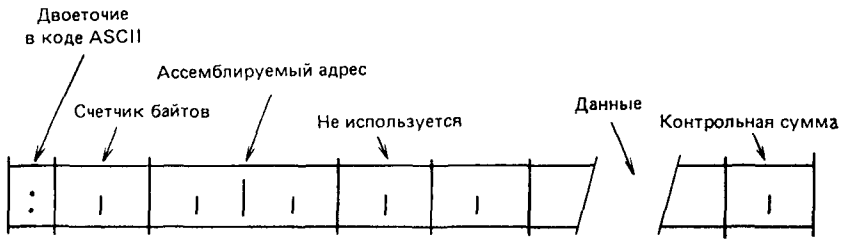

Рис.4.1. Шестнадцатеричный формат объектного кода фирмы Intel

Такой абсолютный загрузчик обычно встроен в операционную систему микрокомпьютора и начинает работать, когда пользователь вводит команду запуска программы.

Наиболее популярный для микрокомпьютора с микропроцессором 8080 формат недвоичного объектного кода был разработан фирмой Intel для систем, использовавших файлы на перфоленте. В шестнадцатеричном формате фирмы Intel каждый байт объектного кода представляется в виде двух символов в коде ASCII: один символ для старшего полубайта, а другой - для младшего[1]. Каждому символу соответствует шестнадцатеричное значение полубайта. Таким образом, байт объектного кода C3 представляется соответственно двумя символами в коде ASCII: C и 3. Файлы в гексадецимальном формате фирмы Intel состоят из записей, формат которых представлен на рис. 4.1.

Каждая запись (или строка) начинается с двоеточия в коде ASCII. Следующие два символа содержат шестнадцатеричное число, указывающее количество байтов в записи (счетчик), при этом учитываются только байты данных (каждый байт данных представляется в виде двух символов). В последней записи файла этот счетчик равен нулю. Следующие четыре символа соответствуют двухбайтовому шестнадцатеричному адресу памяти (старший байт адреса следует первым), присвоенному записи при ассемблировании. По этому адресу находится первый байт данных, а остальные следуют непосредственно за ним. Следующий байт записи текущими версиями гексадецимального загрузчика фирмы Intel не используется. Далее следуют байты данных (по два символа на байт); их число соответствует значению, указанному в поле счетчика. В последнем байте содержится контрольная сумма записи. Он представляет собой дополнение до двух суммы всех байтов данных. Наличие ошибки определяется сравнением этой контрольной суммы с контрольной суммой, вычисленной загрузчиком. Запись заканчивается символами возврата каретки и перевода строки, за которыми могут идти несколько символов нуля.

[1] В отечественной практике этот формат принято называть гексадецимальным. - *Прим.перев.*

Этот формат был разработан главным образом для того, чтобы можно было выявлять ошибки в файлах на перфоленте и чтобы такую информацию можно было прочитать визуально. Однако при таком формате неэффективно используется память на диске, и загрузка по сравнению с загрузкой программы, представленной в двоичном коде, длится дольше.

Для ЭВМ, оснащенных гибкими дисками и имеющих ограниченный объем памяти, двоичный формат был бы более удобным. В различных ассемблерах для двоичных файлов используются различные форматы. Но независимо от формата объектный код и рабочий адрес, присваиваемый коду при ассемблировании, должны совпадать.

После рассмотрения основных элементов языка ассемблера, системы команд микропроцессора 8080 и формата объектного файла нетрудно понять, что делают ассемблеры. То же самое относится и к загрузчику. Загрузчик фирмы Intel, например, просто читает каждую запись, преобразует каждую пару символов в байт, проверяет контрольную сумму и пересылает байты данных в последовательные адреса памяти, начиная с адреса загрузки, заданного в записи.

Загрузив один раз программу, желательно сохранить для будущего использования отображение выполняемой программы в память в виде файла. Во многих операционных системах эта функция выполняется с помощью команды, вводимой с консоли. Казалось бы, разумно иметь ассемблер, генерирующий двоичный объектный код, однако при таком простом подходе теряется гибкость. Часто возникает необходимость ассемблировать программу, а затем исполнять ее в любых заданных адресах памяти. Это особенно важно при работе в мультипрограммном режиме, когда память делится одновременно между несколькими программами. Здесь нельзя заранее сказать, в какой области памяти программа будет работать. В такой ситуации абсолютный загрузчик будет бесполезен. Для этого нужен загрузчик перемещаемых программ, способный загружать объектную программу по любому заданному адресу памяти.

При перемещаемой загрузке необходимо некоторым специальным образом обрабатывать адресные поля загружаемой программы. Если программа ассемблирована для работы начиная с адреса 100, а была загружена начиная с адреса 1000 (адреса шестнадцатеричные), то правильно выполняться она не будет, так как адреса в командах будут отличаться на 0F00. Следовательно, при загрузке программы в память загрузчик должен соответственно изменить каждый адрес. Но для этого загрузчику необходимо сообщить: 1) адрес, для которого программа ассемблировалась, 2) действительный адрес загрузки и 3) какие пары байтов в программе являются адресами. Затем, вычитая 1) из 2), загрузчик получает смещение, которое прибавляет к каждому адресу, содержащемуся в программе, присваивая тем самым адресу правильное значение.

Это смещение может быть отрицательным, если адрес загрузки меньше адреса, полученного в результате ассемблирования, положительным, если адрес загрузки больше, или нулевым, если адреса совпадают.

Как видно из рис.4.1, адрес, для которого ассемблировалась программа, содержится в объектном файле. Действительный адрес загрузки программы может быть задан (или получен) во время загрузки. Не хватает только способа нахождения полей программы, содержащих адреса. Эту функцию выполняет ассемблер: он должен специальным образом помечать адреса в объектном файле. Так что небольшая модификация ассемблера и загрузчика позволит ассемблировать программы в *перемещаемый код* и загружать их в любые заданные области памяти.

Часто желательно разбить большую программу на модули, которые будут разрабатываться и тестироваться по отдельности. Каждый модуль отдельно компилируется, а затем во время загрузки все они объединяются, образуя единую программу. Модули, разработка которых не была закончена, могут заменяться "*заглушками*" - фиктивными или очень простыми модулями, в которых может не содержаться ни одной команды, что позволит проверять уже готовые модули так, как-будто они составляют полную программу. При модульном программировании можно не только разрабатывать по отдельности программные сегменты, но также использовать объектные библиотеки. Получив однажды объектные коды подпрограмм общего назначения, можно обращаться ним в других программах, не вставляя подпрограммы при ссемблировании этих программ.

При таком методе работы необходимо иметь отдельные модули в перемещаемом объектном коде, так как нельзя заранее предсказать, где в памяти будет располагаться каждый модуль. Однако при этом возникает еще одно затруднение: такая программа будет иметь *внешние ссылки*, т.е. адреса в командах, которые ассемблер не может определить, так как эти адреса относятся к операндам или подпрограммам, расположенным вне данного модуля и не известным ассемблеру. У загрузчика появляется дополнительная работа по обнаружению недостающих объектных модулей, их загрузке и организации связи с программой с помощью подстановки недостающих адресов. Такой загрузчик называют *динамическим*[1].

Для своей работы динамический загрузчик должен получить от ассемблера некоторую дополнительную информацию. Объектный файл должен содержать имена всех внешних ссылок. В объектных файлах подпрограмм должны быть описаны имена и положения всех операндов или команд, на которые возможны ссылки из других модулей. Другими словами, должны быть объявлены все точки входа подпрограммы.

[1] В отечественной литературе приняты и другие названия этого загрузчика: программа-сборщик, загрузчик с настройкой по параметрам. - *Прим.перев.*

В объектном файле точка входа определяется ее именем (символьной строкой, содержащей соответствующую метку) и смещением. Смещение - это число, указывающее положение точки входа (в байтах) относительно начала модуля.

Внешние ссылки также описываются с помощью имени и смещения. Однако в этом случае после добавления адреса загрузки модуля смещение указывает на цепочку адресных полей, содержащих одну и ту же внешнюю ссылку. Эти цепочки создаются ассемблером. Так как в процессе ассемблирования невозможно сгенерировать адреса, необходимые для внешних ссылок, ассемблер использует адресные поля для связи всех появлений данной внешней ссылки. В адресное поле первой команды, ссылающейся на данную внешнюю ссылку, записывается нуль, что указывает на конец цепочки. В каждую последующую ссылку записывается смещение относительно предыдущей. В конце программы записываются каждая внешняя ссылка и смещение последнего адресного поля в соответствующей цепочке.

Каждый ассемблер должен уметь различать внешние ссылки, точки входа и обыкновенные метки. Таким образом, необходимо иметь еще две директивы. Для указания того, что метка является внешней ссылкой, используется директива EXT, EXTRN или EXTERNAL. Такие метки не имеют адресов, так как их действительное положение не известно ассемблеру. На месте адреса ассемблер записывает смещение относительно предыдущей ссылки в рассмотренной выше цепочке внешних ссылок. Для объявления метки внешней точкой может быть использована директива ENTRY, GLOBAL или PUBLIC. Так как эти метки находятся внутри ассемблируемой программы, им соответствуют обычные адреса.

Примерами могут служить директивы

```
ABC:  EXTRN
```

(определяет, что метка ABC содержится в другом модуле) и

```
XYZ:  ENTRY
```

(определяет XYZ как обыкновенную метку и, кроме того, объявляет ее точкой входа). В некоторых ассемблерах используется другой синтаксис. Например, те же самые метки могут быть объявлены следующим образом:

```
      EXT   ABC
```

и

```
      ENTRY XYZ
```

В последнем случае метка XYZ должна появиться где-либо в программе в виде обычной метки.

При другом подходе для меток, являющихся точками входа, используются два двоеточия (например, XYZ::), а для внешних ссылок - два символа \\ (например, LHLD ABC\\). Существу-

ют и другие варианты обозначения внешних ссылок и точек входа. Кроме того, некоторые ассемблеры допускают различные методы опередепения таких меток. В одной и той же подпрограмме могут появиться как точки входа, так и внешние ссылки. Получив от программиста всю эту информацию о метках, ассемблер имеет возможность правильно задать точки входа и внешние ссылки в объектном файле. Последняя информация, необходимая динамическому загрузчику, - это имя объектной библиотеки и место (дисковод) ее нахождения. Обычно эти данные задаются загрузчику во время его работы. Однако для задания указанной информации могут использоваться также другие директивы ассемблера и соответственно другие объектные записи.

Альтернативой динамическому загрузчику является *редактор связей*. Он выполняет те же самые операции по организации связи модулей, однако вместо загрузки программы в память записывает ее в файл на диске. Если результатом его работы является двоичное отображение программы, то такой файл представляют собой выполняемую копию программы, и поэтому отпадает необходимость в работе, связанной с загрузкой и сохранением.

До сих пор мы рассматривали программирование на машинном языке и языке ассемблера. Эти языки называют языками программирования *низкого уровня*, так как они непосредственно связаны с работой конкретных процессоров. С разработкой языка ассемблера стало очевидно, что производительность программистов была бы выше, если бы они использовали языки *высокого уровня*, более близкие к обычному языку. На таких языках можно было бы более естественно выражать конкретные типы задач. Они могли бы освободить программмистов от многих мелочей, связанных с программированием на языке ассемблера. Операторы языка высокого уровня могли бы быть более "мощными" и соответствовать нескольким машинным командам.

Эти цели были реализованы с разработкой Фортрана - языка, близкого математическому и предназначенного для решения научных и инженерных задач, и Кобола - языка, близкого естественному английскому и предназначенного для решения коммерческих задач. За ними последовало появление многих других языков высокого уровня. Из них наибольшее распространение получили Алгол, ПЛ/1, Лисп и АПЛ. Позднее были созданы Бейсик, Паскаль и Си. С появлением языков высокого уровня возникло и новое слово для обозначения процесса генерации программ - компиляция. Языки высокого уровня являются *компилируемыми*, а трансляторы с языков высокого уровня называют *компиляторами*.

Теоретически для каждого языка применимы как интерпретация, так и генерация объектного кода. Однако на практике для каждого языка наиболее удобен какой-либо один способ получения объектного кода. Бейсик, например, обычно интерпретируется, хотя имеются и компиляторы Бейсика. Зато Си почти всегда транслируется в машинный код.

ЧАСТЬ 2

ЯЗЫК СМОЛЛ-СИ

В каждой главе этой части рассматривается одна из особенностей языка Смолл-Си. При этом главы можно читать как последовательно для постепенного изучения языка, так и выборочно для получения справочного материала.

Некоторые представленные здесь понятия связаны с длиной слова микропроцессора. Так как было бы затруднительно каждый раз определять понятие длины слова в приложении к каждому из наиболее распространенных микропроцессоров, за основу здесь взята модель 8080 (см. гл.1). Очевидно, что можно использовать данное понятие и применительно к другим микропроцессорам.

В последующих главах рассматривается синтаксис ряда операторов. При этом используются следующие соглашения. Основные термины выделены курсивом и могут связываться дефисами в единые синтаксические единицы. Косая черта / также служит для связи терминов. Ее следует читать как *и/или*. Под словом *строка* понимается ряд символов, написанных подряд без разделяющих пробелов. Слово *список* означает ряд элементов, разделенных запятыми и необязательными пробелами. Многоточием обозначается повторение подобных элементов. Апостроф в конце термина служит для обозначения необязательного элемента. Перед тем как продолжить, будет не лишним познакомиться с языком Смолл-Си, взглянув на пример программы. Программа *words* (см. листинг 5.1) читает по одному слову из входного файла и записывает эти слова в отдельные строки выходного файла. Слово в данном случае - это непрерывная строка, состоящая из печатаемых символов, т.е. символов, имеющих графическое представление.

Первая строка программы является комментарием, в котором указано имя программы и дано краткое описание ее функции. Во второй строке содержится указание включить текст из файла stdio.h. В третьей и четвертой строках определены имена INSIDE и OUTSIDE, соответствующие числам 1 и 0. Встроенный в компилятор препроцессор проверяет каждую строку программы, заменяя эти имена соответствующими им значениями. Это делается до того, как компилятор производит обычную обработку строки. В следующих двух строках определяются переменные - символьная переменная ch и целая переменная where, которой присваивается начальное значение 0 (определенное ранее именем OUTSIDE).

Процедурная часть программы содержит три функции: main, white и black. Выполнение программы начинает функция main, в которой содержатся обращения к функциям white и black. Оператор while управляет повторяемым выполнением следующего за ним оператора if...else... . При каждом повторении происходит обращение к функции getchar для получения следующего символа из входного файла; при этом значение символа присваивается переменной ch. Если возвращаемое функцией getchar значение не равно (!=) значению символа EOF (определенного в файле stdio.h), то данный оператор выполняется; иначе управление возвращается операционной системе. При каждой итерации текущий символ сравнивается с разделителем слов (пробел, переход на новую строку или символ табуляции). Если это разделитель, то вызывается функция white, которая проверяет, находился ли предыдущий символ внутри слова (переменная where имела значение INSIDE). Если это так, то данный символ является первым разделителем после слова, и в этом случае вызывается функция putchar, записывающая в выходной файл символ перехода на новую строку. Затем переменной where присваивается значение OUTSIDE, благодаря чему все следующие разделители данного *слова будут игнорироваться (т.е. в выходной файл после данного* слова не будут записываться символы перехода на новую строку). Как только будет найден печатаемый символ, вызывается функция black, которая записывает этот символ в выходной файл и присваивает переменной where значение INSIDE. Это означает, что последний символ входил в состав слова.

Таким образом, при выполнении этой программы все идущие подряд разделители слов вызывают лишь один переход на новую строку, т.е. в выходной файл или на выходное устройство один раз записывается последовательность символов возврата каретки и перехода на новую строку.

ГЛАВА 5

СТРУКТУРА ПРОГРАММЫ

Как видно из листинга 5.1, программа на языке Си имеет простую структуру. Программа - это, вообще говоря, набор описаний. Существуют описания символьных переменных (например, строка 5 листинга) и целых переменных (строка 6). Ниже будет показано, что можно описывать массивы и указатели[1] как символьных, так и целых переменных. Полная версия языка Си поддерживает и другие типы переменных, в то время как Смолл-Си ограничивается перечисленными выше.

[1] Указатели - это текущие значения адресов. - *Прим.перев.*


```
/*words -- записать каждое слово на отдельную строку */
#include <stdio.h>
#define INSIDE  1
#define OUTSIDE 0
char ch;
int where = OUTSIDE;
main() {
    while((ch = getchar()) != EOF) {
      if((ch == ' ') || (ch == '\n') || (ch == '\t'))
                white();
      else black();
      }
    }
white() {
    if(where == INSIDE) putchar('\n');
    where = OUTSIDE;
    }
black() {
    putchar('\n');
    where = INSIDE;
    }
```

Листинг 5.1. Пример программы на языке Смолл-Си

Затем следуют описания функций. *Функция* - это подпрограмма, вызываемая из различных точек программы. По завершении своей работы функция возвращает управление в ту точку, откуда она вызывалась. Описание функции состоит из двух частей: собственно *описания* и *тела* функции. В описании задаются имя функции и имена передаваемых ей аргументов. В приведенном выше примере программы аргументы не передаются, поэтому во всех трех описаниях функций содержатся пустые списки аргументов (состоящие только из открывающей и закрывающей скобок). Скобки необходимы даже и в том случае, когда аргументы отсутствуют.

В теле функции содержатся описания аргументов, за которыми следуют операторы, заключенные с двух сторон в фигурные скобки. В описаниях аргументов указываются типы аргументов в описании функции. Должен быть описан каждый аргумент; в теле функции допускаются описания только аргументов. Фигурные скобки могут использоваться также для группирования последовательности операторов в *составные операторы*, или *блоки*. Таким образом, оператор в теле функции можно рассматривать как один составной оператор. Когда функция получает управление, работа начинается с первого выполняемого операторы в теле функции. Когда оператор заканчивается, т.е. достигнута закрывающая фигурная скобка, управление возвращается в точку вызова функции, и работа программы продолжается.

Выполнение программы на языке Си начинается с обычного вызова функции main; это значит, что где-либо в программе должна быть функция main. Выход из этой функции (для данного обращения) возвращает управление операционной системе. Некоторые более старые версии Смолл-Си отличались от полной версии Си тем, что выполнение начиналось с вызова первой функции программы, независимо от ее имени. Таким образом, для совместимости всех версий языка Смолл-Си следует начинать программу с функции main.

Как будет показано в гл.14, переменные можно описывать внутри составных операторов. Такие переменные называются *локальными* переменными, так как определены только внутри того составного оператора, в котором появились, и в подчиненных (т.е. внутренних) составных операторах. Переменные, описанные вне функции, например ch и where в примере программы, называются *глобальными*, так как известны всем функциям программы. Их называют также *внешними* переменными, так как они описаны вне границ функций, однако этот термин может внести путаницу, так как он относится и к таким переменным, на которые есть ссылки в одном исходном файле, а сами они описаны в другом. Поэтому название *внешние* будет использоваться только для переменных последнего типа. Функции не могут быть описаны внутри других функций, т.е. функции описываются только на глобальном уровне. Более подробно об описаниях будет рассказано в последующих главах.

Комментарии могут располагаться в любом месте программы. Они ограничиваются слева и справа символами /* и */ соответственно. Длина комментариев не ограничивается, и они могут занимать любое число строк.

Язык Си является языком с произвольным расположением полей. Не имеет значения, в каком месте строки расположен тот или иной символ. Допускаются как строки, содержащие несколько операторов, так и операторы, занимающие несколько строк. Единственное ограничение относится к командам препроцессора (см. гл.15), каждая из которых записывается на отдельной строке.

До сих пор программы на языке Си описывались в терминах одного исходного файла, за исключением случая, когда в заданном месте программы вставлялись строки из другого файла. Однако программы могут состоять из нескольких исходных файлов, компилируемых по отдельности. В компиляторе Смолл-Си предусмотрена возможность объединения этих *подпрограмм* или при ассемблировании, или при загрузке, но не в обоих случаях сразу (см. гл.20).

Если в конфигурации компилятора предусмотрена возможность организации связи подпрограмм во время загрузки, то глобальные переменные из другого файла, на которые есть ссылки в данном файле, должны в данном файле объявляться внешними (см. гл.8). Функции, отсутствующие в исходном файле, содержащем обращения к ним, считаются внешними. Компилятор автоматически объявляет каждый глобальный объект (переменную, массив, указатель или функцию) входной точкой. Однако если предусмотрено, что компилятор сам объединяет части программы, то он работает совершенно иначе и все рассматривает как одну программу.

Таким образом, программа на языке Смолл-Си обычно состоит из одного или нескольких исходных файлов. Связь отдельных частей программы осуществляется тремя способами. Во-первых, исходный текст одного файла может быть включен в другой с помощью оператора #include (см. гл.15). Во-вторых, части программы могут по отдельности компилироваться, а затем ассемблироваться все вместе. При этом требуется, чтобы ассемблер так же, как и компилятор Си, поддерживал функцию включения одного файла в другой. В-третьих, части программы могут компилироваться и ассемблироваться по отдельности, а затем связываться вместе с помощью динамического загрузчика или редактора связей.

Каждый исходный файл обычно содержит команды препроцессора и список глобальных описаний для переменных, массивов, указателей и функций. Каждая функция, в свою очередь, состоит из описания и тела. В описании функции содержатся ее имя и локальные имена получаемых аргументов. Тело функции включает в себя описания аргументов и составной оператор, состоящий из описаний локальных переменных, выполняемых операторов и других составных операторов. В этих составных операторах, в свою очередь, содержатся собственные описания локальных переменных, операторы и составные операторы, и т. д. Выполнение программы, написанной на языке Смолл-Си, начинается с функции main; управление другим функциям передается только в тех случаях, когда к ним есть обращения. Обращение к функции записывается в виде имени функции, за которым следует список из нуля или более аргументов, заключенных в скобки.

ГЛАВА 6

ЭЛЕМЕНТЫ ЯЗЫКА СМОЛЛ-СИ

Возможно, первое, что бросается в глаза при просмотре листинга 5.1, это то, что большая часть программы написана строчными буквами. Все ключевые слова, такие как int, while и if, должны быть написаны строчными буквами. Однако имена, задаваемые пользователем, могут записываться как строчными, так и прописными буквами. Обычно везде, кроме идентификаторов, описываемых в операторе #define, используются строчные буквы.

Наличие прописных букв в таких идентификаторах, как правило, означает, что это имена не переменных, а констант.

Идентификаторы (имена данных и т.п.) могут иметь любую длину, однако значащими являются только первые восемь символов; остальные символы допускаются, но игнорируются. Например, имена nameindex1 и nameindex2 будут рассматриваться компилятором как одно и то же имя nameinde. Имя должно начинаться с буквы, далее могут следовать как буквы, так и цифры. Однако в качестве буквы может использоваться символ подчеркивания _. Таким образом, имена _abc и a_b_c являются допустимыми.

Для каждого глобального имени компилятор Смолл-Си генерирует на языке ассемблера метку с тем же именем. В некоторых ассемблерах длина метки ограничена шестью символами, и не допускается использование специальных символов, а также зарезервированных имен, например названий регистров и директив ассемблера. Поэтому лучше всего такие имена не использовать и делать так, чтобы имена различались по первым шести символам. Кроме того, следует избегать употребления имен, начинающихся с символа подчеркивания или с букв cc, так как они используются в функциях библиотеки нижнего уровня. Это не относится к локальным именам. Они помещаются в стек, и обращение к ним производится не по имени, а по их положению относительно вершины стека.

В языке Си используются следующие знаки пунктуации: точка с запятой, двоеточие, запятая, апостроф, кавычки, а также фигурные, прямые и круглые скобки.

Точка с запятой используется прежде всего как признак конца оператора. Точка с запятой должна стоять в конце каждого простого (несоставного) оператора, даже если он является последним в составном операторе, например

```
{temp = x; x = y; y = temp;}
```

Команды препроцессора являются исключением, так как для каждой из них требуется отдельная строка. Точки с запятой разделяют также три выражения в операторе for (см. гл.14), например

```
for (i = 0; i < 10; i = i + 1) x[i] = 0;
```

Двоеточия ограничивают метки, а также префиксы case и default (эти элементы языка используются в операторах, управление которым передается непосредственно).

Запятыми разделяются элементы списка. Например, три целые переменные могут быть описаны следующим образом:

```
int i, j, k;
```

Обращение к функции, для которой нужно задавать четыре аргумента, может выглядеть так:


```
func (arg1, arg2, arg3, arg4);
```

Запятые, кроме того, используются в качестве разделителей в списках выражений. В следующем примере производится модификация переменных, входящих в один список, и запятая делает этот фрагмент программы более понятным:

```
while (+ +i, - - k) abc ();
```

Апострофы ограничивают символьные константы с обеих сторон. Например 'a' представляет собой константу, равную коду строчной буквы a. Так как в большинстве реализаций языка Си используется набор символов кода ASCII, то значение 'a' равно десятичному числу 97.

Кавычки, по аналогии, ограничивают строки символов, соответствующие символьным массивам.

В круглые скобки заключают составные операторы - блоки операторов, которые выполняются вместе так, как если бы они были одним простым оператором.

В прямые скобки заключают размерности (в описаниях) и индексы (в выражениях) массивов. Таким образом, оператор

```
char string[80];
```

описывает символьный массив string объемом 80 символов, пронумерованных от 0 до 79, а оператор

```
ch = string[79];
```

присваивает последний символ этого массива переменной ch.

В круглые скобки заключают списки аргументов в описаниях функций и в обращениях к ним. Кроме того, эти скобки используют при группировании выражений и подвыражений для управления порядком вычислений.

Как показано в табл.13.1, некоторые специальные символы используются в выражениях в качестве знаков операций. В большинстве случаев знак операции представляется парой символов.

Комментарии в программе на языке Си начинаются с символов /* и заканчиваются символами */. Компилятор игнорирует комментарии, но записывает их в листинг (если требуется).

Таковы элементы языка Си. То, как они используются, станет ясно из следующих глав.

ГЛАВА 7

КОНСТАНТЫ

Компилятор Смолл-Си распознает два типа констант: целые и символьные. Целые константы записываются в виде строки деся-

тичных цифр. Отрицательным значениям предшествует знак минус. Положительные значения могут не иметь знака или же перед ними ставится знак плюс. В большинстве реализаций компилятора Смолл-Си целые константы имеют внутреннее представление в виде 16-разрядного слова со знаком. Это ограничивает диапазоны положительных и отрицательных значений: 0 ... 32767 и -32768 ... -1 соответственно. Следует обратить внимание на то, что отрицательные числа имеют то же самое двоичное представление, что и значения без знака от 32768 до 65535. Компилятор Смолл-Си принимает эти значения без знака и формирует из них отрицательные числа. Однако после того как последние обрабатываются ассемблером, их двоичное представление становится тем же самым, что и для чисел без знака. Следовательно, можно употреблять все числа без знака от 0 до 65535. Однако нужно следить за тем, чтобы при сравнении больших положительных чисел с другими операндами выполнялось именно беззнаковое сравнение. Такое сравнение производится в том случае, когда хотя бы один из сравниваемых операндов является адресом (см. гл.10).

Компилятор Смолл-Си всегда воспринимает целые константы как десятичные. Однако в полной версии языка Си распознаются также восьмеричные и шестнадцатеричные константы. В полной версии языка Си константа, начинающаяся с 0 (нуля), рассматривается как восьмеричная; если константа начинается с 0x или 0X, то она считается шестнадцатеричной. Компилятор Смолл-Си ничего этого не распознает. В первом случае число ошибочно воспринимается как десятичное, а во втором выдается сообщение об ошибке. Поэтому при написании программ на языке Смолл-Си следует избегать использования в числовых константах ведущих нулей, так как если придется когда-нибудь компилировать программу компилятором полной версии языка Си, то могут возникнуть затруднения.

Символьные константы содержат один или два символа, заключенных в апострофы. Может показаться странным, что возможны символьные константы, состоящие из двух символов, однако это имеет смысл, если учесть, что константы (даже символьные) подобно переменным загружаются в виде слов полной длины. Константе 'B' будет соответствовать шестнадцатеричное число 0042 (значение кода ASCII для прописной буквы B), а константе 'AB' - число 4142, т.е. два символа A и B соответственно в старшем и младшем байтах. Заметим, что здесь не происходит распространения знака, как в случае символьных переменных (см. гл.8).

Иногда возникает необходимость задать в программе определенные непечатаемые символы. Для этого можно использовать последовательность из двух или более символов, первый из которых (служебный) изменяет значение остальных. В результате генерируется только один символ. В языке Си роль служебного символа выполняет обратная косая черта \. Компилятор Смолл-

Си распознает следующие последовательности с этим служебным символом:

\n	- новая строка,
\t	- табуляция,
\b	- возврат на шаг,
\f	- управление форматом,
\ooo	- любой символ, представленный в виде восьмеричного числа ooo.

Новая строка - это один символ, который при записи в выходной файл начинает новую строку. При выдаче на экран дисплея происходит перевод курсора в первую позицию следующей строки. При записи на устройство вывода или в символьный файл символ новой строки становится последовательностью, состоящей из двух символов "возврат каретки" и "перевод строки" (не обязательно в приведенном порядке). При вводе происходит обратное преобразование - пара символов "возврат каретки" и "перевод строки" преобразуется в один символ новой строки. В некоторых реализациях компилятора Си символом новой строки служит "возврат каретки", в других - "перевод строки". Однако при этом не возникает проблемы совместимости, поскольку вместо числового значения вы используете последовательность \n.

С помощью последовательности \ooo, состоящей из служебного символа, за которым следует восьмеричное число из одной, двух или трех цифр, можно представить любой символ. Компилятор принимает все цифры в диапазоне 0 ... 7, следующие за служебным символом, пока не наберет три цифры, или же до первого символа, не являющегося восьмеричной цифрой. В полной версии Си есть небольшое отличие: в ней цифры 8 и 9 рассматриваются соответственно как восьмеричные значения 10 и 11.

Существует еще один тип последовательности со служебным символом. Если за обратной косой чертой следует какой-либо символ, отличающийся от описанных выше, то служебный символ игнорируется, а следующий символ рассматривается как константа. Таким образом, написав '\\' можно закодировать как константу обратную косую черту, а с помощью записи '\'' можно задать константу для апострофа.

Строго говоря, компилятор Си не распознает символьные строки, однако он распознает массивы символов и предоставляет программисту способ написания массивов символьных констант, называемых строками. Заключив строку символов в кавычки, вы тем самым определяете массив символов и генерируете адрес этого массива, т.е. в том месте программы, где появляется строка символов, генерируется адрес массива символьных констант, сам же массив располагается в другом месте. Это очень важно запомнить. Отметим, что в этом проявляется отличие от символьной константы, которая прямо генерирует свое значение. В соответствии с соглашением, принятым в языке Си, признаком конца символьной

строки является байт, содержащий нуль. Поэтому компиляторы Си автоматически добавляют этот признак конца символьной строки. Таким образом, в строке

```
"abc"
```

задан массив, состоящий из четырех символов (a, b, c и нуль), при этом генерируется адрес первого символа, используемый программой. Здесь, как и для символьных констант, можно применять последовательность, содержащую служебный символ. Например, кавычки в символьной строке можно обозначить следующим образом:

```
"....\"...."
```

Так как строка может содержать и один, и два символа, то ее можно использовать как альтернативу символьным константам в тех случаях, когда вместо значения символа нужен его адрес.

Заметим, что символьные константы и строки должны записываться целиком в одной строке программы.

ГЛАВА 8

ПЕРЕМЕННЫЕ

В языке Смолл-Си есть переменные двух типов: целые и символьные. Целые переменные занимают в памяти слово, а символьные - байт. Очень важно запомнить следующее: после того как символьная переменная выбирается из памяти, она преобразуется в целую переменную. Сам байт с константой попадает в младшие разряды регистра. Самый левый разряд считается знаковым, и его значение помещается во все разряды старшего байта. Другими словами, символьная переменная становится целой с помощью распространения знакового разряда вдоль старшего байта. Таким образом, символ, имеющий шестнадцатеричное значение 7F (десятичное значение 127), получает шестнадцатеричное значение 007F (десятичное значение остается равным 127), а шестнадцатеричное значение FF (десятичное значение -1) становится равным FFFF (десятичное значение остается равным -1).

Не все компиляторы Си выполняют для символьных переменных распространение знака, поэтому важно отметить, что Смолл-Си это делает, и учитывать возможные эффекты при переносе программ с одного компилятора на другой. Рассмотрим, например, выражение

```
ch < 127
```

где ch - символьная переменная, принимающая значения в диапазоне 0 ... 255. Проблема заключается в том, что все

значения ch, превышающие 127, в действительности делают ее меньшей нуля. Поэтому данное выражение следует переписать следующим образом:

```
(ch & 255) < 127
```

Поразрядная операция И (&) с числом 255 обнуляет старший байт, в то время как младший байт не изменяется. Заметим, что такое выражение будет справедливым для компиляторов обоих типов.

В отличие от выборки из памяти, запись в память производится только для одного символа, для чего необходимо сократить целое до символа, отбросив старший байт. При этом программист должен заботиться о том, чтобы не были утеряны значащие разряды.

Переменная - это операнд, располагающийся в некотором месте памяти. Очень важно различать сам операнд и его адрес. На операнд вы ссылаетесь с помощью его имени, например var. Адрес операнда получается при записи перед знаком операции получения адреса (символ &, который по контексту можно отличить от знака поразрядной операции И). Так, &var есть адрес переменной var.

В отличие от Бейсика и Фортрана, в Си каждая переменная до ее использования должна быть *описана*. *Описание* переменной предполагает выполнение двух шагов: *объявление* ее типа (целая или символьная) и *определение* ее памяти (резервирование места для нее). При описании внешней переменной происходит только присвоение ей имени; действительного опеределения ее не происходит. Полное определение дается в другом исходном файле.

В табл.8.1 приведены примеры описания переменных. Заметим, что описание extern, в котором тип переменной не задан, равносильно заданию типа int. Тот же самый основной синтаксис используется для описания указателей, массивов и внешних функций (см. гл.9, 10 и 12).

Таблица 8.1. **Описания переменных**

Описание	Комментарий
int i;	Определяет переменную i и объявляет ее целой
char x,y;	Определяет переменные x и y и объявляет их символами
extern char z;	Объявляет z символом, описанным в другом месте
extern i, k;	Объявляет i и k целыми, описанными в другом месте

Переменные, описанные на глобальном уровне, называют *статическими* переменными, так как их значения всегда существуют и никогда не теряются, независимо от того, как в программе

передается управление. Область существования глобальной переменной включает в себя всю программу, лежащую ниже описания переменной, т.е. переменная известна всем последующим функциям этой программы. Глобальные имена должны быть в программе уникальными.

В отличие от глобальных, локальные переменные называют *автоматическими*, так как они известны, пока управление передается внутри того блока (составного оператора), в котором эти переменные описаны. Когда управление передается за пределы блока, они исчезают. Локальные переменные являются автоматическими в том смысле, что появляются, когда нужны, и исчезают, когда становятся ненужными. В область существования локальных переменных входят только тот блок, в котором они описаны, и подчиненные ему блоки. Таким образом, можно ссылаться на локальную переменную, описанную в том же блоке или в блоке более высокого уровня, но не в подчиненном блоке. Имя локальной переменной должно быть уникальным только в том блоке, в котором она описана.

Одно и то же имя может быть описано во всех блоках программы. В каждом случае определяются различные переменные. При ссылке на имя поиск описания этого имени происходит снизу вверх, начиная с того уровня, где появилась данная ссылка. Если не найдена такая локальная переменная, то компилятор просматривает имена глобальных переменных. Другими словами, локальные описания "заслоняют" описания более высоких уровней. Это удобно, так как позволяет описывать временно используемые локальные переменные, не заботясь о том, встречаются ли те же самые имена где-либо еще в программе.

В языке Си под словом *объект* понимается любая область памяти, которая может быть изменена. Это общий термин для всего того, на что можно ссылаться и что можно обрабатывать. Все переменные являются объектами, но не все объекты есть переменные. В следующих двух главах рассматриваются указатели и массивы; и те и другие являются объектами. В то же время константы не являются объектами, так как их нельзя изменить.

ГЛАВА 9

УКАЗАТЕЛИ

Одна отличительная особенность языка Си, делающая его удобным для системного программирования, состоит в том, что он позволяет работать с адресами. Это существенно увеличивает гибкость языка и позволяет обращаться из программ, написанных на языке Си, ко всему объему памяти.

Адреса, хранящиеся в памяти подобно обычным переменным, называют *указателями*. Они имеют имена, занимают по одному машинному слову каждый, и работа с ними очень похожа на работу с целыми переменными. Однако когда их сравнивают с другими элементами, то оба сравниваемых элемента рассматриваются как целые положительные без знака. Таким образом, не имеет смысла сравнивать указатель с чем-либо еще, кроме другого адреса. Фактически для сохранения совместимости различных компиляторов языка Си следует сравнивать только адреса внутри массивов. Сравнение любых других адресов предполагает некоторый риск, обусловленный тем, что различные компиляторы могут по-разному организовывать память программы.

Тип указателя связан с объектом, на который он указывает (на целое или символ). В полной версии языка Си указатели могут соответствовать и другим объектам, но в языке Смолл-Си возможны только объекты этих двух типов. Тип указателя важен потому, что разные объекты имеют разную длину. При работе с указателями используются значения адресов объектов, а этими объектами не обязательно являются байты. Прибавление единицы к указателю символа дает указатель на следующий символ, в то время как прибавление единицы к указателю целого дает указатель на следующее целое. Таким образом, любое изменение указателя на какую-либо величину должно быть промасштабировано компилятором для учета того, что целое занимает в памяти больше одного байта.

Некоторые адреса не являются указателями потому, что не имеют имен или их нельзя модифицировать. Первый случай относится к адресам символьных строк (см. гл.7), а второй - к адресам массивов (см. гл.10). В данной главе рассматриваются только такие адреса, которые являются указателями.

При описании указателей применяется тот же самый синтаксис, что и при описании переменных (см. гл.8), с тем лишь отличием, что перед именами указателей ставятся звездочки. Фактически указатели и переменные могут присутствовать в одном и том же описании. Например, описание

```
int i, *ip;
```

определяет целое i и указатель целого ip. Звездочка здесь читается как слово *объект*. Идея состоит в том, что и целое i, и объект, на который указывает указатель ip, являются целыми. Подобным же образом описание

```
char *cp;
```

определяет cp и объявляет его указателем на символьный объект. Заметим, что cp занимает целое слово, так как это указатель, а не символ.

Если указатель появляется слева от знака операции присваивания или ему предшествует знак операции увеличения или умень-

шения, то значение указателя изменяется. Если он появляется где-либо в выражении, то его значение (обычно адрес) выбирается из памяти и используется таким, какое оно есть. Таким образом, с указателями можно работать точно так же, как с одиночными переменными. Единственное отличие состоит в том, что при сравнении указатели считаются беззнаковыми целыми числами, а увеличение или уменьшение указателя масштабируется в соответствии с его типом (более подробно адресная арифметика описана в гл.10).

Если перед указателем стоит знак операции обращения по адресу *, то читается из памяти или записывается в память объект того же самого типа, что и указатель. Точнее говоря, сначала указатель загружается в регистр, а затем из регистра получают адрес для операции загрузки или записи в память. Звездочку называют *знаком операции обращения по адресу*, потому что ссылка на объект происходит косвенно, и при этом сначала получается адрес объекта.

Все вышеизложенное лучше пояснить на примере. Рассмотрим фрагмент программы (листинг 9.1), в котором к соответствующим целым числам добавляются пять символов. Прежде всего указатель cend устанавливается на пять символов дальше некоторого адреса, хранящегося в указателе cp. Затем оператор while многократно проверяет, выполняется ли условие cp < cend. Если это так, то выполняется составной оператор; в противном случае управление передается следующему оператору программы (отсутствующему в этом фрагменте). При каждом выполнении составного оператора объект с адресом cp (символ) прибавляется к объекту с адресом ip (целому). Затем указатели cp и ip увеличиваются так, чтобы они указывали на следующие объекты. Так как ip - указатель целого, то каждое увеличение переставляет его вперед не на 1, а на 2 байта. Эта процедура повторяется пять раз.

```
char *cp, *cend;
int  *ip;
.
.
.
cend = cp + 5;
while (cp < cend)  {
   *ip = *ip + *cp
   ip = ip + 1;
   cp = cp + 1;
   }
```

Листинг 9.1. Пример использования указателей

ГЛАВА 10

МАССИВЫ

В языке Смолл-Си допускаются только одномерные массивы. Они организуются в виде последовательности целых или символьных переменных, называемых элементами массива. Число элементов задается в описании массива выражением, следующим непосредственно за именем массива. Это выражение заключается в квадратные скобки и может включать в себя только константы. Примеры правильного описания массивов приведены в табл.10.1.

Таблица 10.1. Описания массивов

Описание	Комментарий
int ia1[25], ia2[SZ + 1];	Описывает массивы целых чисел: ia1, состоящий из 25 целых, и ia2, состоящий из (SZ + 1) целых. (Значение SZ должно быть определено как константа или константное выражение.)
extern int ia[];	Описывает массив целых чисел, размерность которого задана вне функции.
char ca[8];	Описывает массив ca, состоящий из восьми символов.

Пример, в котором не задана размерность массива, может вызвать недоумение. Как компилятор может работать с массивом неопределенной длины? Здесь подчеркивается одна важная особенность языка Си. Размерности в языке Си служат только для того, чтобы знать, какой объем памяти необходимо зарезервировать; ответственность за установку границ массивов целиком ложится на программиста. Поэтому компилятору совсем не нужно знать размерности массивов, описанных как внешние объекты или являющихся аргументами функции, так как эти массивы определены где-то в другом месте.

Имя массива указывает на адрес первого элемента массива. Так как массивы фиксированы в памяти, то их адреса неизменны. Поэтому имена массивов можно использовать в качестве адресов, но нельзя присваивать именам массивов новые значения. Однако элементы массивов можно изменять.

Ссылка на элемент массива записывается в виде имени массива, за которым следует выражение для индекса, заключенное в квадратные скобки. В качестве индекса массива может быть использовано любое допустимое выражение (см. гл.13). Индекс 0 соответствует первому элементу, индекс 1 - второму, и т. д.

Таким образом, первому и последнему элементам массива ca соответствуют обозначения ca[0] и ca[7].

В гл.8 рассказывалось, как для получения адреса применяется знак операции получения адреса &. Эту же операцию можно использовать для получения адреса массива. Так, выражение

```
&ca[3]
```

дает адрес четвертого элемента массива ca. Заметим, что выражения

```
&ca[0]
```

и

```
ca + 0
```

и

```
ca
```

эквивалентны. По аналогии ясно, что выражения

```
&ca[3]
```

и

```
ca + 3
```

также эквивалентны.

Чтобы обратиться к элементу массива, компилятор Смолл-Си добавляет к адресу массива индекс (соответствующим образом увеличенный для целых чисел). В результате получается адрес объекта, который необходимо выбрать из памяти или записать в память. Программист может сослаться на элемент массива и иначе - прибавив индекс к имени массива и использовав для получения результата операцию обращения по адресу. Таким образом, выражения

```
ca[x]
```

и

```
*(ca + x)
```

эквивалентны. Заметим, что круглые скобки здесь обязательны, так как последнее выражение будет эквивалентно следующему:

```
(*ca) + x
```

Первый элемент массива ca может быть записан как

```
ca[0]
```

или

```
*(ca + 0)
```

или даже как


```
*ca
```

Из сказанного выше следует, что в некоторых случаях могут использоваться как указатели, так и имена массивов. Указатели могут иметь индексы, а имена массивов могут использоваться в качестве адресов. Таким образом, если предположить, что в указателе на целоео ip содержится адрес массива целых, можно будет обратиться к пятому элементу одним из следующих способов:

```
ip[4]
```

или

```
*(ip + 4)
```

Единственным ограничением для полной взаимозаменяемости имен массивов и указателей является то, что имена всегда соответствуют фиксированным адресам: они хранятся в памяти не как переменные и, следовательно, их нельзя изменять. В соответствии с этим оператор

```
ia = x;
```

(где ia - имя массива целых) будет неверным, а оператор

```
*ia = x;
```

- верным, так как он изменяет объект с адресом ia, а не сам адрес ia.

Как вы уже видели, адреса (указатели и имена массивов) могут свободно использоваться в выражениях. Однако смысл имеют только две операции: изменение адреса на некоторую величину и получение разности двух адресов. Все другие возможные операции дают неопределенные результаты.

Изменение указателя или имени массива в положительном направлении имеет смысл. Изменение же имени в отрицательном направлении бессмысленно, так как в результате получается адрес памяти вне этого массива. В то же время указатели не привязаны к началу массива, поэтому их отрицательное изменение может быть полезным.

Разность двух адресов дает число элементов, лежащих между этими адресами. Например, выражение

```
ip1 - ip2
```

(где ip1 и ip2 - указатели целых) дает число целых между этими адресами. Компилятор генерирует команды для вычитания ip2 из ip1, а затем делит результат на число байтов в целом. Для указателей символов деление не производится. Отсюда ясно, что можно написать и бессмысленные операции, такие как получение разности адреса символа и целого (результатом чего является разность адресов, не относящихся к одному и тому же массиву)

или же вычитание большего адреса из меньшего. Компилятор допускает такие операции, однако пользы от этого не будет.

И последний момент, относящийся к адресной арифметике: компилятор Смолл-Си не позволяет использовать целые без знака. Однако вы можете описать указатель символов и обращаться с ним, как с целым без знака. Совсем не обязательно, чтобы значение указателя в действительности было адресом памяти. Однако вы должны следить за использованием указателя символов, так как автоматическое масштабирование при сложении с указателем на целое (или вычитании из него) может привести к нежелательному эффекту.

Пример, приведенный в листинге 9.1, можно переписать, применяя массивы. Результат показан в листинге 10.1. Сначала устанавливается в нуль целая переменная i. Следующий за этим оператор while передает управление составному оператору, выполняющему основную работу. До тех пор пока i меньше пяти, соответствующие элементы массивов ca и ia складываются и результат попадает в ia. При каждой итерации i увеличивается на единицу. Когда i становится равным пяти, управление передается оператору, следующему за оператором while.

```
char ca[5];
int  ia[5], i;
.
.
.
i = 0;
while (i < 5)  {
    ia[i] = ia[i] + ca[i];
    i = i + 1;
    }
```

Листинг **10.1**. Пример использования массивов

ГЛАВА 11

НАЧАЛЬНЫЕ ЗНАЧЕНИЯ

В полной версии языка Си можно присваивать начальные значения как глобальным, так и локальным объектам, в то время как в языке Смолл-Си инициализируются только глобальные объекты. Локальные объекты всегда имеют неопределенные начальные значения. Начальные значения глобальных объектов всегда известны, если они не заданы, то считаются равными нулю. Связанные с этим преимущества приводят и к некоторым ограничениям, на

одном из которых следует остановиться. Но сначала о преимуществах.

Присваивая глобальным объектам начальные значения, вы тем самым избавляетесь от необходимости писать операторы присваивания, выполняющие эту работу. Для переменных и указателей выгода здесь небольшая, однако для массивов разница более существенная, так как для них приходится писать операторы организации циклов или же целый список операторов присваивания. Размер объектной программы, в которой используется задание начальных значений, несколько меньше из-за отсутствия операторов присваивания. Однако на скорости выполнения программы это существенно не сказывается, так как операторы присваивания начальных значений выполняются только один раз.

Недостатком, связанным с присваиванием начальных значений, является то, что теряется возможность *повторно выполнять* программу без ее перезагрузки. Некоторые операционные системы позволяют повторно запускать программы после их первого выполнения. Это особенно удобно при использовании гибких дисков так как для загрузки с этих дисков требуется довольно значительное время, особенно если программа находится на диске, которого в данный момент нет в дисководе. Чтобы программу можно было запускать несколько раз подряд, ее предыдущее выполнение не должно влиять на ход последующих. Это значит, что при каждом выполнении программы начальные значения переменных не должны отличаться. Если возможность такого последовательного использования программы для вас важна, то для задания начальных значений переменных, изменяемых в процессе выполнения программы, необходимо применять операторы присваивания. Тогда при каждом последовательном выполнении начальные значения этих переменных будут присваиваться заново.

Метод задания начальных значений прост: в описании объекта после его имени необходимо задать знак равенства и константное выражение для требуемого значения. При этом допускаются также символьные константы со служебным символом (обратной косой чертой). Таким образом, оператор

```
int i = 80;
```

описывает i как целое и задает ему начальное значение 80, а оператор

```
char ch = '\n';
```

описывает ch как символ и задает начальное значение, равное символу перевода строки. Если необходимо инициализировать массив переменных, то записывается заключенный в фигурные скобки список переменных, разделенных запятыми. Так, оператор

```
int ia[3] = {1, 2, 3};
```

описывает массив целых ia и присваивает его элементам значения соответственно 1, 2 и 3. Если размер массива не задан, то он определяется числом начальных значений. Таким образом, оператор

```
char ca[] = {'a', 0};
```

описывает массив символов ca, состоящий из двух элементов, которым присваиваются начальные значения строчной буквы a в коде ASCII и нуль. Если размер массива превышает число начальных значений, то первые элементы массива инициализируются, а остальные элементы принимаются равными нулю. Следовательно, оператор

```
int ia[3] = 1;
```

описывает массив целых ia, состоящий из трех элементов, первому из которых присваивается значение 1, а остальным - нуль. Если число начальных значений превышает заданный размер массива, то компилятор генерирует сообщение об ошибке.

Для символьных массивов и указателей символов начальные значения можно задавать с помощью символьной строки, заключенной в кавычки. При этом автоматически генерируется нуль в конце этой строки (признак конца). Таким образом, оператор

```
char ca[4] = "abc"
```

описывает символьный массив ca, состоящий из четырех элементов, первым трем из которых присваиваются значения a, b и c, соответственно, а четвертому - нуль. Если размер массива не задан, то он устанавливается как размер строки плюс 1. Следовательно, в описании

```
char ca[] = "abc"
```

массив ca также состоит из четырех элементов. Если размер задан и строка короче массива, то остальные элементы устанавливаются равными нулю. Таким образом, массив, описанный как

```
char ca[6] = "abc"
```

содержит нули в последних трех элементах. Если же строка длиннее, то размер массива соответственно увеличивается.

Когда инициализируется указатель символов, то он устанавливается на адрес строки символов, содержащей начальные значения. Таким образом, оператор

```
char *cp = "name"
```

описывает cp как указатель символов, содержащий адрес пятисимвольной строки name, за которой следует нуль.

В табл.11.1 приведены разрешенные комбинации типов объектов и начальных значений. Указанные здесь типы совместимы с полной версией языка Си, имеющей более широкие возможности.

Таблица 11.1. Разрешенные комбинации объект - начальное значение

Объект	Начальное значение		
	Константное выражение	Список константных выражений	Символьная строка
Символьная переменная	Y		
Указатель на символ		Y	Y
Символьный массив	Y	Y	Y
Целая переменная	Y		
Указатель на целое			
Массив целых	Y	Y	

ГЛАВА 12

ФУНКЦИИ

Подпрограммы в языке Си реализуются в виде *функций*, названных так по аналогии с математическими функциями. Функция может быть *вызвана* в любом месте выражения. Для этого необходимо написать ее имя и заключенный в круглые скобки список из нуля или более аргументов (параметров), передаваемых функции. Например, в выражении

```
func (a,b) + 1
```

вызывается функция func и ей передаются аргументы a и b. Функция всегда возвращает некоторое значение, которое может быть использовано в последующих вычислениях выражения. В примере, приведенном выше, для получения окончательного результата выражения значение, возвращаемое функцией func, прибавляется к 1. Функции, которые не возвращают явно какого-либо значения, возвращают тем не менее некоторое непредсказуемое значение. Это вполне приемлемо, так как часто функции вызываются не для того, чтобы задать в выражении какое-либо значение, а для других действий в программе или при ее взаимодействии с внешним миром.

Рассмотрим сначала, как описывается функция, а затем - как она вызывается. Есть два способа описания функции. Во-первых, если она находится в другом исходном файле и не включается в программу с помощью команды #include (см. гл.15), то ее имя

может быть определено в описании extern. Таким образом, можно явно описать функцию abc как внешнюю, написав

```
extern int abc();
```

То же самое можно записать следующим образом:

```
extern int (*abc)();
```

Смысл этой записи состоит в том, что abc является указателем функции, а *abc (по аналогии с операцией обращения по адресу (см. гл.13) - объектом в abc, т.е. собственно функцией (хотя, строго говоря, функция не является объектом). В полной версии языка Си эти два описания различаются, однако в языке Смолл-Си оба случая трактуются одинаково, при этом фактический указатель функции не генерируется.

Функции языка Смолл-Си всегда возвращают целые значения; поэтому внешние функции могут появляться только в описаниях extern int. Напомним, однако, что если тип в описании extern не задан, то принимается тип int.

В последних версиях языка Смолл-Си (начиная с версии 2.1) не требуется, чтобы внешние функции описывались явно. В этих версиях для каждой неописанной функции, на которую есть ссылка, в программе автоматически генерируются необходимые команды. Поэтому в действительности нет надобности задавать для функций описание extern.

Второй способ задания функций состоит в полном их описании. Такое описание имеет вид

имя (список-аргументов') описание-аргумента'...
{описание-объекта'... оператор'...}

Имя, круглые и фигурные скобки являются необходимыми элементами описания. *Имя* - это имя функции. Оно должно отвечать требованиям к именам языка Си, описанным в гл.6.

Список-аргументов' - список из нулевого или большего числа разделенных запятыми имен параметров, которые будут переданы функции при ее вызове. Это имена аргументов внутри функции. Они называются *формальными параметрами*, в отличие от *фактических параметров*, передаваемых функции при ее вызове.

Описание-аргумента'... - набор описаний, обозначающих атрибуты аргументов, перечисленных в списке. Каждое имя из *списка-аргументов* должно быть описано с помощью описания char или int.

Звездочка перед именем или квадратные скобки после имени задают определенный вид аргумента. Таким образом, описания

```
int *arg;
```

и

```
int arg[];
```


эквивалентны. Размерности массивов в описании аргументов игнорируются компилятором, так как при компиляции в функции не используются размеры передаваемых ей массивов. Размерности могут быть получены с помощью других аргументов или каким-либо иным способом.

Остальную часть описания функции можно рассматривать как один составной оператор, содержащий описания локальных переменных, выполняемые операторы и подчиненные составные операторы.

Рассмотрим в качестве примера следующее описание функции:

```
func (i, c, ia, cp)
    int i, ia[];
    char c, *cp;
    {...}
```

Функция func получает четыре аргумента: целое, символ, массив целых и указатель символов. Многоточием здесь обозначены операторы, составляющие тело функции; они будут рассмотрены ниже (см. гл.14). Отметим, что описывается каждый аргумент, а между списком аргументов и составным оператором описываются только аргументы. Порядок описания аргументов не имеет значения. Функцию без аргументов следует описывать так:

```
func c () {...}
```

Ранее отмечалось, что вызов функции появляется в выражениях. Выражение может ничего не содержать, кроме вызова функции, и так как одиночный оператор является правильным выражением (см. гл.14), вызов функции может быть записан как полный оператор. Таким образом, оператор

```
func c (x, y + 1, 33);
```

мог бы вызвать функцию func и передать ей три аргумента. Так как в этом операторе-выражении нет операции присваивания, значение, возвращаемое функцией func, игнорируется. Оператор

```
x = func ();
```

вызывает функцию func без аргументов и присваивает переменной x возвращаемое значение.

Иногда желательно вызвать функцию и передать ей имя другой функции, которую первая будет вызывать. Например, если func1 и func2 являются функциями, то оператор

```
func1 (func2);
```

вызывает функцию func1, передавая ей адрес func2. Функция func2 предварительно должна быть описана как функция либо явно (одним из приведенных выше способов), либо неявно (при вызове неописанной функции). При описании функции, являю-

щейся формальным параметром, и вызове этой функции следует использовать следующий синтаксис:

```
func (arg) int (*arg) ();{
    ...
    (*arg) ()
    ...
    }
```

Здесь применяется синтаксис для указателей, так как аргумент фактически является указателем на функцию. Этот синтаксис согласуется с полной версией языка Си и был введен в Смолл-Си, начиная с версии 2.1. В более ранних версиях для описания таких аргументов допускался оператор

```
int arg;
```

а для вызова функции использовалось выражение

```
arg (...)
```

Другая особенность старых версий языка Смолл-Си состоит в том, что для задания адресов функций в выражениях можно писать только имена функций без последующих круглых скобок. В этих случаях функция не вызывается, но ее адрес присутствует в выражении. Современный компилятор по-прежнему допускает подобные отклонения, однако для совместимости с полной версией языка Си их следует избегать.

Вызов функции имеет вид

первичное-значение (список-выражений)

где *первичное-значение* - операнд выражения или выражение в круглых скобках. В полной версии языка Си требуется, чтобы в *первичном-значении* так или иначе фигурировало имя функции, но в языке Смолл-Си допускается любое выражение. В то время как компилятор полной версии языка Смолл-Си не допускает такого, например, вызова функции

```
256 ()
```

компилятор Смолл-Си воспринимает это как вызов функции по адресу **256** (десятичное значение адреса).

Теперь рассмотрим подробнее вопросы, связанные с передачей аргументов. В языках программирования используются два способа передачи аргументов: *вызов по ссылке* и *вызов по значению*. При *вызове по ссылке* аргументы передаются таким образом, что ссылки на формальные параметры становятся ссылками на фактические параметры. При такой системе операторы присваивания оказывают неявное побочное влияние на фактические параметры; это значит, что на переменные, переданные функции, оказывает влияние изменение формальных параметров, описанных внутри функции. Иногда такое изменение оказывается полезным, но чаще

- нет. При таком подходе часто возникает необходимость описывать локальные переменные, используемые в качестве рабочих ячеек для аргументов, чтобы функция не изменяла фактических параметров.

В языке Си используется *вызов по значению*, при котором для каждого фактического параметра (любого допустимого выражения) создается и передается функции временный объект. При ссылке внутри функции на формальные параметры происходит ссылка на эти временные объекты. Их можно изменять, ни о чем не заботясь, причем на фактические параметры это не будет оказывать влияния, что облегчает работу программиста и исключает необходимость описывать много формальных параметров.

Однако и при таком способе передачи аргументов возможны побочные эффекты, так как в качестве аргументов могут передаваться адреса. *Объекты*, находящиеся по этим адресам, могут быть изменены при выполнении операции обращения по адресу. Другой способ передачи аргументов состоит в индексировании аргументов, являющихся адресами. Напомним, что для функции и аргументы-указатели, и аргументы-массивы являются указателями, а указатели могут иметь индексы подобно тому, как если бы они были именами массивов. Еще один способ передачи аргументов состоит в использовании в качестве аргументов глобальных переменных.

Каждый фактический параметр является выражением, передающим функции при ее вызове одно слово. Выражения для аргументов вычисляются слева направо. Так как в выражения могут входить операции присваивания, увеличения и уменьшения на 1 (см. гл.13), то они могут влиять на значения аргументов, находящихся справа от знака операции. Этого следует избегать, так как описание языка Си не определяет порядка вычисления аргументов и любая зависимость от порядка вычислений, применяемого конкретным компилятором, будет создавать несовместимость с другими компиляторами.

Фактические параметры записываются в стек в том порядке, в котором они вычисляются. Затем команда CALL записывает в стек выше фактических параметров адрес возврата. Команда RET, возвращающая управление программе, вызвавшей функцию, читает из стека адрес возврата и счетчик команд. После возврата из функции все фактические параметры утрачиваются. Переменные, объявленные в функции локальными, также помещаются в стек (выше адреса возврата). Когда блок, в котором они объявлены, заканчивает свою работу, занятое ими в стеке место освобождается. Следовательно, при каждом вызове сохраняются аргументы, адрес возврата и локальные переменные текущего вызова, и, когда вызванная ранее функция вызывает еще одну функцию, путаницы не происходит. При увеличении вложенности вызовов функции длина стека растет, а при обратном процессе - уменьшается. Так как память ограничена, то всегда существует воз-

можность переполнения стека. Если стек выйдет за пределы отведенной для него области, то вероятнее всего это приведет к нарушению работы системы. Глубину вложенности функций должен контролировать программист. В библиотеке Смолл-Си (см. гл. 17) есть функция avail, проверяющая условие переполнения стека.

Компилятор не следит за тем, чтобы число и тип фактических параметров при вызове функции соответствовали числу и типу формальных параметров в описании функции. Если задано слишком много аргументов, то функция рассматривает только последние их них. Однако если аргументов меньше, чем ожидается, то функция будет использовать элементы стека, лежащие ниже списка аргументов, пока не получит требуемое число аргументов, что наверняка приведет к непредсказуемому поведению программы. Вы любой ценой должны избегать этого.

В отличие от полной версии Си, в Смолл-Си есть средства, с помощью которых функция может определить, сколько аргументов было передано ей в действительности. При каждом вызове функции ей в регистре ЦП передается число аргументов, которое может быть получено данной функцией с помощью обращения к специальной функции CCARGC и присваивания возвращенного значения какой-либо переменной. Эта функция является частью исполнительной системы компилятора Смолл-Си и в самой программе не описывается. Вызов функции CCARGC должен быть первым выполняемым оператором функции, так как при работе регистры ЦП изменяются и число аргументов может быть утрачено. Получить его можно с помощью оператора

```
count = CCARGC();
```

Передача числа аргументов связана с некоторыми накладными расходами, поэтому из программ, в которых не вызываются функции, требующие подсчета числа аргументов, ее можно исключить. Когда компилятор обнаруживает команду

```
#define NOCCARGC
```

он не генерирует кодов для передачи числа аргументов (более подробно команда #define рассматривается в гл.15). Для функций *printf*, *fprintf*, *scanf* и *fscanf* библиотеки Смолл-Си требуется знать число аргументов, поэтому в программах, вызывающих эти функции, вы никогда не должны определять имя NOCCARGC.

Описанный выше метод обращения к функциям обладает тем преимуществом, что допускает *рекурсивные вызовы*. Рекурсивный вызов состоит в том, что функция либо прямо вызывает себя, либо обращается к другим функциям, которые прямо или косвенно опять вызывают ее. Использование рекурсивных вызовов может упростить многие алгоритмы, однако при этом увеличивается необходимый размер стека, и логика программы обычно становится менее понятной. В качестве примера рассмотрим функцию display, приведенную в листинге 12.1. Эта функция выдает в обратном

порядке символы какой-либо строки. При первом вызове в качестве аргумента она получает строку символов (в действительности - адрес первого элемента массива символов). Если символ по этому адресу не равен нулю, то функция вызывает сама себя, передавая адрес следующего элемента массива. Так будет продолжаться до тех пор, пока не будет обнаружен нулевой символ и восстановлено предыдущее состояние функции. В этом случае управление передается в точку вызова функции putchar, которая записывает текущий символ в стандартный выходной файл (см. гл.16). При этом происходит возврат из функции после предыдущего вызова, и т. д., пока управление не будет возвращено вызывающей программе. В качестве упражнения вы можете переписать эту функцию без использования рекурсии.

```
display (string) char string[]; {
    if (*string) {
        display (string + 1);
        putchar (*string);
        }
    }
```

Листинг **12.1**. *Пример рекурсивного вызова функции*

Существует два механизма возврата управления из функции в вызывающую программу. Когда управление при работе функции достигает самой правой фигурной скобки, происходит неявный возврат. В этом случае возвращаемое значение не определено. В вызывающей программе значение функции нельзя использовать, так как оно не определено. Явный возврат может быть задан с помощью оператора

return *список-выражений*;

где **return** - необходимое ключевое слово, а *список-выражений* - произвольный список выражений. Если задано выражение (или список выражений), то вызвавшей программе возвращается его значение (или значение последнего выражения списка); в противном случае возвращаемое значение не определено.

ГЛАВА 13

ВЫРАЖЕНИЯ

Обычно в большинстве языков программирования выражением считается некоторая комбинация констант, переменных, элементов массивов и вызовов функций, объединенных с помощью различных знаков математических операций, дающая в результате одно числовое значение. В языке Си это понятие расширено с помо-

щью введения других типов данных и других (нематематических) операций. Допускаются указатели, имена массивов без индексов и имена функций и, как видно из табл.13.1, обширный набор операций. Наряду со стандартными математическими операциями здесь есть также логические операции, операции отношений, операции сдвига, операции получения адреса и обращения по адресу, операции увеличения и уменьшения на единицу, операции присваивания. Из всех этих операций можно получить любую комбинацию, составляющую выражение. В результате язык Си позволяет писать более компактные и эффективные выражения, которые на первый взгляд могут показаться несколько странными. Однако перед тем как рассматривать все виды выражений, которые могут быть написаны на языке Си, мы обсудим процесс вычисления выражений и некоторые общие свойства операций.

Основной вопрос, возникающий при вычислении выражений, состоит в том, что необходимо определить соответствие между отдельными частями выражения и операциями. Чтобы исключить неопределенность, операциям приписывают три характеристики: *число операндов*, *старшинство* и *ассоциативность*.

В зависимости от того, над каким числом операндов выполняются операции - одним или двумя, их делят соответственно на *унарные* и *бинарные*. Например, унарный знак минус меняет знак следующего за ним операнда на противоположный, в то время как бинарный знак минус служит для вычитания одного операнда из другого.

Старшинство операций позволяет определить, как объединять операнды со знаками операций. Например, при вычислении выражения

```
a + b * c
```

сначала получают произведение b и c, а затем к результату прибавляют a. Этот порядок следует из того, что операция умножения по старшинству стоит выше операции сложения. Для изменения нормального порядка операций при вычислении выражения или для того, чтобы сделать нормальный порядок операций более наглядным, используются круглые скобки. Вычисление такого выражения начинается с самых внутренних скобок, затем продолжается для охватывающих их скобок, и т. д. Каждое выражение в скобках дает один операнд. Следовательно, переписав приведенный выше пример как

```
(a + b) * c
```

мы изменим нормальный порядок вычислений, в то время как выражение

```
a + (b * c)
```

делает нормальный порядок операций более наглядным.

В табл.13.1 приведены операции языка Смолл-Си по убыванию старшинства сверху вниз сначала в левой, а затем в правой колонках. Операции, перечисленные вместе, равны по старшинству и выполняются в порядке появления (исключение составляет случай, когда используются скобки).

Последняя характеристика - ассоциативность определяет направление вычисления последовательности операций данного типа: слева направо или справа налево. Эту характеристику называют также группированием, так как вычисление каждой группы операндов (или одного операнда) дает одно значение, которое становится затем или операндом для следующей операции, или значением всего выражения. Следовательно, при вычислении выражения

```
a + b + c
```

слагаемые группируют слева направо и сначала складывают a и b, а затем к полученной сумме прибавляют c. В табл.13.1 направление группирования для различных операций показано стрелками.

В языке Смолл-Си в результате выполнения каждой операции получается целое значение, которое может соответствовать числу, коду символа, значению *истина*/*ложь*, адресу или чему-либо еще по вашему желанию. Как показано в гл.8, при ссылке на символьные переменные они удлиняются до целых с помощью распространия знака. В гл.7 указывается, что символьные константы могут состоять из одного или двух символов; в первом случае в старший байт записывается нуль. Когда значения выражений присваиваются символьным переменным, старший байт отбрасывается.

Функции в Смолл-Си всегда возвращают целые значения. Если функция не возвращает значение явно, то она дает неопределенный результат. Это может не вызвать каких-либо затруднений, так как совсем не обязательно, чтобы значение, возвращаемое функцией, где-либо использовалось. Однако, если функция сама является операндом большего выражения, она обязательно должна возвращать некоторое определенное значение.

Рассмотрим каждую из этих операций. В последующих пояснениях слово *операнд* относится или к значению одного объекта, или к промежуточному значению, являющемуся результатом частичного вычисления выражения. Таким образом, в выражении

```
(a + b) * c
```

для операции + в качестве операндов используются a и b, в то время как для операции * одним операндом является (a + b), а другим - c.

Если при вычислении выражения встречается неопределенное имя, то оно неявно объявляется именем функции. Если за ним следуют круглые скобки, то происходит вызов функции; в противном случае используется ее адрес. В полной версии языка Си неопределенное имя является функцией только в том случае, если

ним следуют круглые скобки, т.е. если оно записано как вызов ункции.

Таблица 13.1. Операции языка Смолл-Си

Логическое НЕ <--	== Равно -->
Дополнение до единицы	!= Не равно
+ Увеличение на единицу	---
- Уменьшение на единицу	& Поразрядное И -->
Унарный минус	---
Обращение по адресу (объект в)	^ Поразрядное исключающее ИЛИ -->
Определение адреса (указатель на)	---
---	| Поразрядное ИЛИ -->
Умножение -->	---
Деление	&& Логическое И -->
Деление по модулю (остаток)	---
---	|| Логическое ИЛИ -->
Сложение -->	---
Вычитание	= Присваивание <--
---	+= Сложение и присваивание
: Сдвиг влево -->	-= Вычитание и присваивание
· Сдвиг вправо	*= Умножение и присваивание
---	/= Деление и присваивание
Меньше, чем -->	%= Деление по модулю и присваивание
= Меньше или равно	&= Поразрядное И и присваивание
Больше, чем	|= Поразрядное ИЛИ и присваивание
= Больше или равно	^= Поразрядное исключающее ИЛИ и присваивание
	<<= Сдвиг влево и присваивание
	>>= Сдвиг вправо и присваивание

МАТЕМАТИЧЕСКИЕ ОПЕРАЦИИ

+ Сложение

Операция сложения выполняет алгебраическое сложение двух соседних операндов и дает в результате их сумму. Операнды группируются слева направо. Если один операнд является адресом, а второй им не является, то второй операнд воспринимается как смещение на некоторое число объектов (символов или целых в

зависимости от типа адреса). Следовательно, перед сложением он умножается на число байтов в объекте. Для символьных адресов смещение не изменяется, так как каждый символ состоит из 1 байта. Результат, получаемый при сложении адреса и смещения, рассматривается как адрес того же самого типа.

- Вычитание

Бинарная операция вычитания выполняет вычитание правого операнда из левого и дает в результате их разность. Операнды группируются слева направо. Если один операнд является адресом, а второй им не является, то второй операнд воспринимается как смещение на некоторое число объектов (символов или целых в зависимости от типа адреса). Следовательно, перед сложением он умножается на число байтов в объекте. Для символьных адресов смещение не изменяется, так как каждый символ состоит из 1 байта. Результат, получаемый при вычитании адреса и смещения, рассматривается как адрес того же самого типа.

Если оба операнда являются адресами одного и того же типа, то результат корректируется таким образом, чтобы представить число объектов, лежащих между ними (не адрес). Он делится на число байтов, содержащихся в объекте. Если в вычитании участвуют два символьных адреса, то результат не изменяется. Если адреса не одного и того же типа, или первый адрес меньше второго, или оба адреса не относятся к одному и тому же массиву, то результат, скорее всего, будет бесполезен.

* Умножение

Операция умножения дает в результате произведение двух соседних операндов. Операнды группируются слева направо.

/ Деление

Операция деления дает в результате частное от деления левого операнда на правый. Операнды группируются слева направо.

% Деление по модулю

Операция деления по модулю дает в результате остаток от деления левого операнда на правый. Операнды группируются слева направо.

- Унарный минус

Операция унарный минус изменяет на противоположный знак операнда, находящегося справа от знака операции. От бинарной операции вычитания эту операцию отличают по контексту, так как слева от знака операции нет операнда. Операнды группируются справа налево.

ЛОГИЧЕСКИЕ ОПЕРАЦИИ

Существуют только две логические константы: *истина* и *ложь*. Любой операнд может быть проверен логически: нуль соответствует значению *ложь*, а любое ненулевое значение соответствует значению *истина*. Логические операции проверяют логические значения и генерируют логические значения. Они всегда дают единицу для значения *истина* и нуль для значения *ложь*. Это следует иметь в виду при чтении последующих описаний операций.

! Логическое НЕ

Эта унарная операция дает в результате логическое отрицание операнда, стоящего справа от знака операции. Если значение операнда *ложь*, то операция дает значение *истина*; в противном случае операция дает значение *ложь*. Операнды группируются справа налево.

&& Логическое И

Эта бинарная операция дает в результате логическое И для соседних операндов. Если оба операнда имеют значение *истина*, то операция дает значение *истина*; иначе операция дает значение *ложь*. Операнды группируются слева направо. Если в выражении ряд таких операций, то они выполняются одна за другой до тех пор, пока одна из них не даст значение *ложь*. Так как это значение определяет окончательный результат, то для всех оставшихся операций генерируется значение ложь и они не выполняются. Этот прием характерен и для полной версии языка Си, поэтому им можно пользоваться, не опасаясь возможной несовместимости. Для повышения эффективности подобных сложных проверок их можно писать таким образом, чтобы при работе программы вероятность выполнения последних операций была наименьшей, или же в последних операциях следует проверять данные, которые уже участвовали в предыдущих проверках.

|| Логическое ИЛИ

Эта бинарная операция дает в результате логическое ИЛИ для соседних операндов. Если хотя бы один операнд имеет значение *истина*, то и результатом операции будет значение *истина*; иначе - значение *ложь*. Операнды группируются слева направо. Если в выражение входит ряд таких операций, то они выполняются одна за другой до тех пор, пока одна из них не даст значение *истина*. Так как это значение определяет окончательный результат, то для всех оставшихся операций генерируется значение *истина* и они не выполняются. Этот прием характерен и для полной версии языка Си, поэтому им можно пользоваться, не опасаясь возможной несовместимости.

ОПЕРАЦИИ ОТНОШЕНИЯ

Все операции отношения выполняют числовое сравнение двух операндов и дают в результате логическое значение *истина* (единица) или *ложь* (нуль) в зависимости от того, удовлетворяют ли операнды заданному отношению. Если ни один из операндов не является адресом, то происходит алгебраическое сравнение: оба операнда рассматриваются как числа со знаком. Если хотя бы один из операндов является адресом, то оба они рассматриваются как положительные числа без знака. Операнды группируются слева направо.

< Меньше, чем

Эта операция дает в результате значение *истина*, если левый операнд меньше, чем правый; иначе она дает значение *ложь*.

<= Меньше или равно

Эта операция дает в результате значение *истина*, если левый операнд меньше, чем правый, или равен ему; иначе она дает значение *ложь*.

> Больше, чем

Эта операция дает в результате значение *истина*, если левый операнд больше, чем правый; иначе она дает значение *ложь*.

>= Больше или равно

Эта операция дает в результате значение *истина*, если левый операнд больше, чем правый, или равен ему; иначе она дает значение *ложь*.

== Равно

Эта операция дает в результате значение *истина*, если два операнда равны друг другу; иначе она дает значение *ложь*.

!= Не равно

Эта операция дает в результате значение *истина*, если два операнда не равны друг другу; иначе она дает значение *ложь*.

ПОРАЗРЯДНЫЕ ОПЕРАЦИИ

Эти операции выполняют логические операции над отдельными разрядами операндов. Бинарные операции (&, ^ и |) проверяют соответствующие разряды двух операндов для определения соответствующих разрядов результата. Другими словами, младшие

разряды каждого операнда проверяются для определения младшего разряда результата, и т. д. для всех остальных разрядов.

~ Дополнение до единицы

Эта унарная операция дополняет разряды операнда таким образом, что каждый единичный разряд становится нулевым, и наоборот. Операнды группируются справа налево.

& Поразрядное И

Эта операция дает в результате поразрядное логическое И для соседних операндов. Если соответствующие разряды *обоих* операндов равны единице, то соответствующий разряд результата также равен единице; иначе он равен нулю. Операнды группируются слева направо.

| Поразрядное ИЛИ

Эта операция дает в результате поразрядное логическое ИЛИ для соседних операндов. Если *один* (или *оба*) из соответствующих разрядов операндов равен единице, то соответствующий разряд результата равен единице; иначе он равен нулю. Операция является *включающей* в том смысле, что она включает и тот случай, когда оба соответствующих разряда равны единице. Операнды группируются слева направо.

^ Поразрядное исключающее ИЛИ

Эта операция дает в результате поразрядное логическое исключающее ИЛИ для соседних операндов. Если *один* (но *не оба*) из соответствующих разрядов операндов равен единице, то соответствующий разряд результата равен единице; иначе он равен нулю. Операция является *исключающей* в том смысле, что она исключает случай, когда оба соответствующих разряда равны единице. Операнды группируются слева направо.

ОПЕРАЦИИ СДВИГА

Результат выполнения этих бинарных операций равен левому операнду, сдвинутому арифметически влево или вправо на число позиций, указанное в правом операнде. При каждом сдвиге в младший разряд записывается нуль.

<< Сдвиг влево

Эта операция дает в результате левый операнд, сдвинутый влево на число позиций, указанное в правом операнде. При каждом сдвиге в младший разряд записывается нуль.

>> Сдвиг вправо

Эта операция дает результате левый операнд, сдвинутый вправо на число позиций, указанное в правом операнде. При каждом сдвиге знаковый разряд не изменяется и распространяется в следующий младший разряд.

ОПЕРАЦИИ ПРИСВАИВАНИЯ

Каждая из этих бинарных операций присваивает операнду, стоящему слева от знака операции, новое значение. Во всех операциях присваивания операнды группируются справа налево. Новое значение - это или правый операнд, или значение, получаемое из левого и правого операндов. Операнд, которому присваивается значение, называется lvalue. Если это символьный объект, то в него пересылается только младший байт присваиваемого значения. Некоторые операнды, например имена массивов без индексов, вычисляемые выражения, перед которыми нет знака операции обращения по адресу *, и имена, перед которыми стоит знак операции получения адреса &, не являются правильными операндами lvalue, так как в действительности для них не резервируется место в памяти. И наоборот, допустимыми операндами lvalue в Смолл-Си являются только имена переменных, имена указателей с индексами и без индексов, имена массивов с индексами и выражения, перед которыми стоит знак операции обращения по адресу.

Все операции присваивания, кроме одной, имеют формат ?=, где первый символ (в данном случае ?) отличается для разных операций присваивания. Операции присваивания допускают стенографический формат записи

```
a = a ? b
```

Таким образом, выражение

```
a += b
```

эквивалентно выражению

```
a = a + b
```

а выражение

```
a <<= b
```

эквивалентно выражению

```
a = a << b
```

Такие операции присваивания дают более эффективный объектный код, так как операнд а вычисляется только один раз.

В начальной реализации операций присваивания в полной версии языка Си знак равенства стоял в начале, а не в конце. В

результате некоторые знаки операций (= +, = -, = * и = &) допускали двойное толкование.

В языке Смолл-Си, в отличие от полной версии языка Си, подобные сочетания всегда рассматриваются как пары знаков операций. Поэтому для совместимости с полной версией всегда ставьте пробел между знаками операций в подобных парах.

Так как присваивание представляет собой часть вычисления выражения, то традиционные операторы присваивания в действительности являются просто самостоятельными выражениями. А так как все операции дают в результате выражения, которые могут быть использованы в процессе вычисления выражения, в выражении может фигурировать любое число операций присваивания. Чаще всего многократное присваивание выглядит следующим образом:

```
a = b = c = 5
```

Так как в этих операциях операнды группируются справа налево, сначала значение 5 присваивается переменной c, а затем выполняется самая правая операция, дающая присваиваемое значение, после чего в результате выполнения средней операции это значение присваивается переменной b, что дает операнд для следующей операции. В заключение самая левая операция присваивает значение этого операнда переменной a. Операции выполняются так, как если бы выражение было переписано следующим образом:

```
a = (b = (c = 5))
```

Выражение

```
val[i] = i = 5
```

заслуживает отдельного рассмотрения. Заметим, что переменная i модифицируется справа от первого знака операции присваивания и используется также слева от него в качестве индекса. В языке Смолл-Си в качестве индекса будет использовано первоначальное значение i, а в полной версии языка Си - модифицированное значение. Поэтому для совместимости с полной версией языка Си следует избегать подобных выражений.

К операциям присваивания относятся следующие операции:

= Присвоить

Эта операция присваивает значение правого операнда левому операнду.

+= Сложить и присвоить

Эта операция присваивает сумму левого и правого операндов левому операнду.

-= Вычесть и присвоить

Эта операция вычитает правый операнд из левого и присваивает результат левому операнду.

***= Умножить и присвоить**

Эта операция присваивает произведение левого и правого операндов левому операнду.

/= Разделить и присвоить

Эта операция делит левый операнд на правый и присваивает частное левому операнду.

%= Разделить по модулю и присвоить

Эта операция делит левый операнд на правый и присваивает остаток левому операнду.

&= Выполнить операцию поразрядное И и присвоить

Эта операция присваивает левому операнду результат операции поразрядное И для левого и правого операндов.

|= Выполнить операцию поразрядное ИЛИ и присвоить

Эта операция присваивает левому операнду результат операции поразрядное ИЛИ для левого и правого операндов.

^= Выполнить операцию поразрядное исключающее ИЛИ и присвоить

Эта операция присваивает левому операнду результат операции поразрядное исключающее ИЛИ для левого и правого операндов.

<<= Сдвинуть влево и присвоить

Эта операция арифметически сдвигает левый операнд влево на число разрядов, указанное в правом операнде. Результат присваивается левому операнду.

>>= Сдвинуть вправо и присвоить

Эта операция арифметически сдвигает левый операнд вправо на число разрядов, указанное в правом операнде. Результат присваивается левому операнду.

ОПЕРАЦИИ УВЕЛИЧЕНИЯ И УМЕНЬШЕНИЯ НА ЕДИНИЦУ

++ Увеличить на единицу

Эта унарная операция увеличивает операнд. Если знак операции стоит перед операндом, то результатом является увеличенное

значение, если же после операнда, то результатом является исходное значение, но сам операнд при этом увеличивается. Операнды группируются справа налево. Если операнд - адрес, то эта операция увеличивает его до адреса следующего символа или целого в зависимости от типа адреса; иначе операнд увеличивается на единицу.

-- Уменьшить на единицу

Эта унарная операция уменьшает операнд. Если знак операции стоит перед операндом, то результатом является уменьшенное значение, если же после операнда, то результатом является исходное значение, но сам операнд при этом уменьшается. Операнды группируются справа налево. Если операнд - адрес, то эта операция уменьшает его до адреса предыдущего символа или целого в зависимости от типа адреса; иначе операнд уменьшается на единицу.

ОПЕРАЦИИ ПОЛУЧЕНИЯ АДРЕСА И ОБРАЩЕНИЯ ПО АДРЕСУ

& Получение адреса

Эта унарная операция дает в результате адрес операнда, перед которым стоит знак операции. В Смолл-Си операция получения адреса не группирует операнды; ее можно применять только непосредственно к переменной, указателю (с индексом или без) или имени массива.

* Обращение по адресу

Знак этой унарной операции ставится перед адресным операндом, изменяя его значение с *адреса* на *объект по адресу*. Если операндом является адрес целого, то операция обращения по адресу обеспечивает ссылку на целое по этому адресу. Если операндом является адрес символа, то операция обращения по адресу обеспечивает ссылку на символ по этому адресу. Операнды группируются справа налево.

ГЛАВА 14

ОПЕРАТОРЫ

Операторы появляются только внутри функций. Они выполняются последовательно в том порядке, в котором записаны. Терминатором (признаком конца) оператора является символ "точка с запятой"; терминатор ставится после каждого простого (не составного) оператора. Некоторые операторы управляют выполнением других операторов. В подобных случаях терминатор выполняемого оператора является терминатором и всего комплекса. Если последний оператор является составным оператором, то терминатором для него служит его самая правая ограничивающая скобка. Внутри составного оператора управление передается последовательного от одного оператора к другому. Ниже описаны все операторы языка Смолл-Си.

ПУСТЫЕ ОПЕРАТОРЫ

Простейшим оператором языка Си является пустой оператор. Он состоит только из терминатора оператора, т.е. из точки с запятой. Как и следует из названия этого оператора, он ничего не делает. Однако он может быть полезен вместе с управляющим оператором. Соответствующий пример приведен ниже в описании оператора while.

СОСТАВНЫЕ ОПЕРАТОРЫ

Операторы могут быть сгруппированы в *составные операторы* или блоки. Их можно использовать везде, где синтаксис языка позволяет помещать простые операторы. Составные операторы имеют вид

{описание-объекта'...оператор'...}

Ограничивающие составной оператор фигурные скобки необходимы. *Описание-объекта'...* - набор из нуля или более локальных описаний. *Оператор'...* - набор из нуля или более простых или составных операторов. Первый элемент (т.е. описание) должен предшествовать последнему.

Когда управление передается в составной оператор, выполняются два действия. Сначала для локальных переменных, если они есть, выделяется пространство в стеке. Затем обрабатываются выполняемые операторы. Когда заканчивается выполнение составного оператора, пространство, занимаемое локальными переменными, освобождается.

Для локальных переменных не могут быть заданы начальные значения; значения локальных переменных до их задания всегда не определены. Для инициализации локальных переменных нужны операторы присваивания.

Область действия локальных переменных включает тот блок, в котором они определены, и распространяется вниз (в иерархии подчинения) на все подчиненные блоки. В блоках, расположенных выше, эти переменные "не видны". Описание локальных переменных "заслоняет" синонимы, описанные на более высоком уровне. Это означает, что для операторов, попадающих в область действия двух или более переменных с одинаковым именем, всегда "видны" только локальные описания самого низкого уровня.

Важное правило, связанное с реализацией компилятора Смолл-Си, состоит в том, что вход в блок, содержащий описания локальных переменных, должен всегда происходить через его левую фигурную скобку. Операторы goto и switch (см. ниже) дают возможность прямо передать управление заданному оператору. Так как они нарушают это правило, компилятор Смолл-Си запрещает использование оператора goto в функциях, содержащих описания на уровнях ниже тела функции, а также запрещает описания внутри операторов switch. В полной версии языка Си подобных ограничений нет.

ОПЕРАТОРЫ-ВЫРАЖЕНИЯ

Везде, где синтаксис языка допускает использование выражений, может появиться список выражений, разделенных запятыми. В таких случаях выражения вычисляются одно за другим слева направо, при этом самое правое выражение определяет значение этого списка. Любое выражение (или список выражений) можно рассматривать как самостоятельный оператор. В обычных условиях значение такого выражения (или списка) не используется. Оно загружается в регистр ЦП, но затем игнорируется. Такие значения можно использовать, вводя команды на языке ассемблера после этого оператора (подробности приведены в гл. 15). Операторы-выражения полезны тем, что обычно в них производится вычисление выражений. При вычислении выражений выполняются вызов функций, операции присваивания, увеличения и уменьшения на единицу. Вот несколько примеров таких операторов:

```
func ();
++i;
i = 15, j = 16, k = 17;
x += func (x, y = 12) * 100;
```

ОПЕРАТОР GOTO

Оператор goto прерывает нормальное последовательное выполнение операторов, передавая управление прямо в заданную точку. Он имеет формат

goto *имя*;

где *имя* - метка, которая должна находиться в той же самой функции, что и оператор goto. В функции такая метка должна быть уникальной. Метки имеют формат

имя:

Имя подчиняется правилам образования имен в языке Си (см. гл. 6). Заметим, что метки заканчиваются двоеточием. Этим подчеркивается тот факт, что они не являются операторами, но,

предшествуя операторам, отмечают точки, существенные для логики программы. Оператор goto передает управление прямо оператору, следующему за меткой.

Напомним, что оператор goto нельзя использовать в функциях, в которых локальные переменные были описаны на уровнях ниже, чем тело функции. Таким образом, если в функции содержится оператор goto, то все локальные переменные должны быть описаны в самом начале.

ОПЕРАТОР IF

Оператор if может иметь один из двух форматов:

if (*список-выражений*) *оператор*
if (*список-выражений*) *оператор* **else** *оператор*

Список-выражений - одно или более выражений, разделенных запятыми; *оператор* - любой простой или составной оператор. В первую очередь вычисляется и проверяется *список-выражений*. Если задано больше одного выражения, то проверяется значение, которое дает самое правое выражение. Если это значение *истина* (т.е. не нуль), то выполняется первый (или единственный) оператор, а оператор, следующий за ключевым словом else (если оно есть), пропускается. Если же это значение *ложь* (т.е. нуль), то первый оператор пропускается, а второй (если он есть) выполняется. Например, оператор

```
if (ch) {
        putchar (ch);
        ++i;
        }
else return (i);
```

проверяет символьную переменную ch. Если она не нулевая, то функция putchar записывает ее в стандартный файл вывода, и переменная i увеличивается; в противном случае функция, в которую входит данный оператор, возвращает значение i.

ОПЕРАТОР SWITCH

Оператор switch проверяет выражение на равенство одному из заданных значений. Затем выполняется оператор, выбранный в зависимости от значения выражения. Оператор switch имеет формат

switch (*список-выражений*) {*оператор'...*}

где *список-выражений* - список, состоящий из одного или нескольких выражений; *оператор'...* - операторы, которые должны быть выбраны для выполнения. Они выбираются с помощью префикса case или default - специальных меток, используемых только

в операторе switch. Эти префиксы указывают на точки внутри составного оператора, на который передается управление в зависимости от значения *списка-выражений*. Для оператора switch они выполняют ту же роль, что и метки для оператора goto, т.е. указывают на точки, в которые может быть передано управление. Префикс case имеет формат

case *константное-выражение*:

а префикс default имеет формат

default:

Двоеточие в конце префикса аналогично двоеточию в конце метки простого оператора. Выражение в префиксе case может содержать только константы и знаки операций.

После вычисления списка-выражений ищется первый совпадающий префикс case. Затем управление передается прямо в эту точку, в результате чего выполняется следующий за префиксом оператор. Если был выбран один из префиксов case, то остальные префиксы case и префикс default не оказывают никакого влияния; они пропускаются, как будто их вообще нет. Если не найдено ни одного совпадающего префикса case, управление передается на префикс default, если он есть. При отсутствии префикса default весь составной оператор игнорируется и управление передается дальше. В операторе switch может появиться только один префикс default. В полной версии языка Си не допускается наличия нескольких префиксов case с одинаковым значением, в то время как в языке Смолл-Си выбирается первый из них.

Если необходимо, чтобы после выполнения выбранного префикса произошел выход из оператора switch, т.е. чтобы остальная часть составного оператора не выполнялась, можно использовать оператор break.

Он имеет формат

break;

Вышесказанное можно пояснить несколькими примерами. Оператор

```
switch (ch) {
   case 'Y':
   case 'y': cptr = "yes";
                        break;
   case 'N':
   case 'n': cptr = "no";
                        break;
   default: cptr = "error";
   }
```

проверяет символьную переменную ch. Если она равна 'Y' или 'y', то символьный указатель cptr устанавливается на адрес строки, содержащей "yes", и выполнение оператора заканчивается. Если она равна 'N' или 'n', то указатель cptr устанавливается на адрес строки, содержащей "no", и выполнение оператора switch также заканчивается. В противном случае указатель cptr устанавливается на адрес строки, содержащей "error", и выполнение передается за пределы оператора switch через конец составного оператора.

Оператор

```
switch (i) {
   default: putchar ('a');
   case 2: putchar ('b');
   case 3: putchar ('c');
   }
```

сравнивает переменную i со значениями 2 и 3. Если переменная i не равна ни одному из них, то в стандартный файл вывода записываются символы abc. Если i = 2, то записываются символы bc, если i = 3, то записывается только символ c.

Тело оператора switch не является обычным составным оператором, так как в нем и в подчиненных ему блоках не допускаются локальные описания. Это вытекает из правила языка Смолл-Си, по которому вход в блоки, содержащие описания, может осуществляться только через открывающую фигурную скобку.

ОПЕРАТОР WHILE

Оператор while управляет повторением выполнения других операторов. Он имеет формат

while (*список-выражений*) *оператор*

где *список-выражений* - список из одного или нескольких выражений, а *оператор* - простой или составной оператор. Если задано больше одного выражения, то проверяемое значение дает самое правое из них. Сначала вычисляется *список-выражений*. Если он дает значение *истина* (не нуль), выполняется *оператор*, и *список-выражений* вычисляется снова. До тех пор, пока *список-выражений* дает значение *истина*, повторяется выполнение *оператора*. Когда значение *списка-выражений* становится равным значению *ложь*, *оператор* пропускается, и управление передается дальше.

В примере

```
i = 5;
while (i) array[--i] = 0;
```

обнуляются элементы с 0 по 4 массива array. Сначала значение переменной i устанавливается равным 5. До тех пор, пока значение переменной i не будет равно нулю, выполняется оператор присваивания. Каждый раз перед тем, как значение переменной i

будет использовано в качестве индекса для массива array, оно уменьшается.

Часто можно встретить такой вид оператора while, в котором управляемый оператор является пустым оператором, и вся работа выполняется на фазе проверки. Примером может служить оператор

```
while (*dest++ = *sour++);
```

в котором dest и sour являются указателями. При каждой итерации программа получает объект, находящийся по адресу sour, до увеличения указателя sour. Затем, до увеличения указателя dest, это значение присваивается объекту, находящемуся по адресу dest. Так как операция присваивания дает в результате присваиваемое значение, то оно становится и значением выражения в скобках. Если оно не равно нулю, то выполняется пустой оператор и призводится следующее вычисление. Этот процесс повторяется до тех пор, пока не будет присвоено значение нуль. Этот пример служит иллюстрацией того, почему в языке Си строки заканчиваются нулевым байтом.

Для управления выполнением составных операторов удобно вместе с оператором while использовать два других оператора. Оператор continue имеет формат

```
continue;
```

и вызывает передачу управления сразу в начало оператора while для следующего шага вычислений. Если оператор while (или какие-либо другие комбинации операторов управления циклом) вложены друг в друга, то оператор continue оказывает влияние только на выполнение самого внутреннего оператора.

Оператор break (описанный ранее) вызывает передачу управления за пределы оператора while. Если операторы while (или switch, или какие-либо другие комбинации операторов для управления циклом) вложены друг в друга, то оператор break оказывает влияние только на выполнение самого внутреннего оператора.

Нет ничего необычного в таком операторе while:

```
while (1) {
    ...
    break;
    ...
    }
```

где многоточия заменяют простые последовательности операторов. Заметим, что здесь выражение всегда имеет значение *истина*. Обычный способ выхода из такого цикла - воспользоваться оператором break. Для того чтобы избежать бесконечного выполнения цикла, можно закончить выполнение программы, обратившись к функции exit (см. гл.17).

ОПЕРАТОР FOR

Оператор for также управляет циклами в программе. Он представляет собой улучшенный вариант оператора while, в котором управление переменными цикла осуществляется тремя операциями (инициализировать, проверить и модифицировать), связанными между собой синтаксически. Этот оператор имеет формат

for (*список-выражений*;
список-выражений;
список-выражений;) *оператор*

Оператор for выполняется следующим образом:

1. Первый *список-выражений* вычисляется только один раз.
2. Второй *список-выражений* вычисляется для того, чтобы определить, следует ли выполнять *оператор*. Если задано более одного выражения, то проверяемое значение дает самое правое из них. Если оно дает значение *ложь* (нуль), то управление передается за пределы оператора for. Если оно дает значение *истина* (не нуль), то *оператор* выполняется.
3. Вычисляется третий *список-выражений*, и процесс возвращается к шагу 2.

Приведенный ранее пример, в котором массив из пяти элементов заполняется нулями, с использованием оператора for может быть переписан следующим образом:

```
for (i = 4; i >= 0; --i) array [--i] = 0;
```

или еще лучше так:

```
for (i = 5; i; array [--i] = 0);
```

Любой из трех списков выражений оператора for может быть опущен, однако разделяющие точки с запятыми при этом должны оставаться. Если отсутствует проверяющее выражение, то результат всегда *истина*. Таким образом, оператор

```
for (;;) {...break;...}
```

будет выполняться до тех пор, пока не встретится оператор break.

Так же как и в операторе while, здесь могут использоваться операторы break и continue, выполняющие те же самые функции, что и в операторе while. Оператор break вызывает передачу управления за пределы оператора for. По оператору continue вычисляется третий *список-выражений*. Другими словами, оператор continue выполняет функцию передачи управления прямо в конец составного оператора, управляемого оператором for.

ОПЕРАТОР DO/WHILE

Оператор do/while - это такой оператор while, в котором внутренний *оператор* выполняется до вычисления *списка-выражений*. Он имеет формат

do *оператор* **while** (*список-выражений*);

где *оператор* - любой простой или составной оператор. Оператор do/while выполняется следующим образом:

1. Выполняется *оператор*.

2. Вычисляется и проверяется *список-выражений*. Если задано больше одного выражения, то проверяемое значение дает самое правое из них. Если оно дает значение *истина* (не нуль), то управление возвращается к шагу 1; в противном случае управление передается за пределы оператора do/while.

Так же как и в операторах while и for, здесь могут быть использованы операторы break и continue. В этом случае оператор continue вызывает передачу управления прямо на вычисление *списка-выражений*. Оператор break вызывает передачу управления за пределы оператора do/while.

ОПЕРАТОР RETURN

Оператор return используется внутри функции для возврата управления в точку, из которой функция была вызвана. Оператор return не всегда необходим, так как достижение конца функции всегда подразумевает возврат. Однако он необходим, если требуется вернуться из какого-либо другого места функции или нужно вернуть вызывающей программе какое-либо значение. Оператор return имеет формат

return *список-выражений*';

где *список-выражений* - необязательный список выражений. Если *список-выражений*' указан, то значение, возвращаемое функцией, определяется последним выражением; если он отсутствует, то возвращаемое значение не определено.

ЗАБЫТЫЕ ОПЕРАТОРЫ?

Вы, вероятно, заметили, что не были описаны операторы ввода-вывода, управления программой и памятью. Это сделано потому, что в языке Си такие операторы отсутствуют. Дело в том, что подобные функции обязательно зависят от реализации языка. Чтобы сделать язык в некоторой степени машинно-независимым, был определен набор стандартных функций. Стандартные функции, используемые при работе с компилятором Смолл-Си, описаны в гл.17.

ГЛАВА 15

КОМАНДЫ ПРЕПРОЦЕССОРА

Для внесения в исходный файл различного рода изменений при компиляции компиляторы производят препроцессирование.

Препроцессор позволяет применять простые макроопределения, условную компиляцию и включение текста из других файлов. В зависимости от реализации препроцессор может быть отдельной программой или может быть встроен в сам компилятор. В любом случае препроцессор обрабатывает команды, синтаксис которых не является частью языка. Каждая команда записывается в отдельной строке и начинается с символа #. Такие команды рассматриваются препроцессором, а не компилятором. В Смолл-Си различие между препроцессором и компилятором несколько размыто, однако для лучшего понимания работы компилятора команды, начинающиеся символом #, можно рассматривать как команды препроцессора.

МАКРООПРЕДЕЛЕНИЯ

Команды, имеющие формат

#define *имя символьная-строка*'

могут быть использованы для определения символов, которые стоят в произвольных строках текста. *Имя* должно удовлетворять принятым в языке Си соглашениям по заданию имен. *Символьная-строка*' начинается с первого печатаемого символа и продолжается до конца строки или начала комментариев. Эти команды должны появляться на глобальном уровне. После команды #define каждое *имя* (за исключением строк констант и символьных констант) заменяется на *символьную-строку*'. Если *символьная-строка*' отсутствует, то соответствующее имя просто удаляется из текста. Проверяется полное имя (до восьми символов); совпадение части имени не принимается во внимание.

Таким образом, команда

```
#define ABC 10
```

заменит

```
i = ABC;
```

на

```
i = 10;
```

но никак не повлияет на

```
i = ABCD;
```

Обычно для макроимен применяют прописные буквы, чтобы они по внешнему виду отличались от переменных. Замены выполняются и для последующих команд #define, так что новые имена могут быть определены в терминах, предшествующих им имен.

В большинстве случаев команды #define используются для присваивания константам имен. Однако вы можете заменять имя

и еще чем-нибудь, например часто встречающимся выражением или последовательностью операторов.

УСЛОВНАЯ КОМПИЛЯЦИЯ

Условная компиляция позволяет обозначать части программы, которые могут компилироваться или не компилироваться в зависимости от значения предварительно определенных признаков. Используя команды условной компиляции, можно писать процедуры, которые включаются в программу или исключаются из нее простым изменением команд #define в начале программы.

Когда препроцессор встречает команду

#ifdef *имя*

он проверяет, было ли определено заданное имя. Если оно не было определено, то препроцессор пропускает все последующие строки, пока не встретит команду

#else

или

#endif

Команда #endif ограничивает ту часть текста, которая управляется командой #ifdef. Команда #else позволяет разделить этот текст на истинную и ложную части. Первая часть (#ifdef... #else) компилируется только в том случае, если заданное имя было определено, а вторая (#else... #endif) - если нет.

Обратной по смыслу команде #ifdef является команда

#ifndef *имя*

Она также действует совместно с командами #else и #endif. В этом случае, если заданное имя не было определено, компилируется первая (#ifndef... #else) или единственная (#ifndef... #endif) часть текста; в противном случае компилируется вторая часть (#else... #endif), если, конечно, она есть.

Допускается вложенность этих команд. На глубину вложенности не накладывается никаких ограничений, кроме размера области памяти, предназначенной во время компиляции для стека. Можно, например, написать следующее:

```
#ifdef ABC
    ...                 /* ABC */
    #ifndef DEF
    ...                 /* ABC и не DEF */
    #else
    ...                 /* ABC  и DEF */
    #endif
    ...                 /* ABC */
```



```
#else
    ...                 /* не ABC */
    #ifdef HIJ
    ...                 /* не ABC, но HIJ */
    #endif
    ...                 /* не ABC */
#endif
```

где многоточия соответствуют какому-либо коду языка Си, а комментарии указывают, при каких условиях будут компилироваться различные части кода.

ВКЛЮЧЕНИЕ ДРУГИХ ИСХОДНЫХ ФАЙЛОВ

Препроцессор распознает также команды включения в процесс компиляции исходного кода из других файлов. Команды

#include "*имя-файла*"

и

#include <*имя-файла*>

и

#include *имя-файла*

задают исходный файл для компилятора. Препроцессор заменяет эти команды текстом из заданного файла, и в результате компилятор воспринимает только этот текст. После того как весь файл прочитан, продолжается обычная обработка последующего текста.

Формат *имени-файла* соответствует формату спецификации файла в используемой операционной системе. В полной версии языка Си требуются кавычки или угловые скобки, а в Смолл-Си - нет. В любом случае, чтобы включить стандартный файл определений для ввода-вывода (этот файл содержит стандартные определения и обычно включается в каждую программу на языке Смолл-Си), для обеспечения совместимости следует писать

#include <stdio.h>

а для других файлов -

#include "*имя-файла*"

Эта команда позволяет использовать во многих различных программах набор часто применяемых функций. Она облегчает разработку программ и обеспечивает единообразие различных программ. Вместо включения общих функций во время компиляции можно то же самое сделать (с помощью соответствующих ассемблера и загрузчика) во время загрузки без перекомпиляции общих функций вместе с каждой программой. (Более подробно это описано в гл.4.)

Для большинства программистов одним из главных оснований для использования языка Си является достижение переносимости[1] программ. Но иногда возникают ситуации, когда для выполнения каких-либо интерфейсных функций (или в каких-либо других случаях) необходимо иметь полный доступ к средствам операционной системы (или аппаратным средствам). Если такие случаи встречаются очень редко и не повторяются во многих программах, то их отрицательное влияние на переносимость программ может быть вполне допустимым.

В подобных ситуациях компилятор Смолл-Си предоставляет возможность вставлять команды на языке ассемблера в нужные места программы на языке Си. Так как выходом компилятора является программа на языке ассемблера, то эти команды просто копируются прямо в выходной файл. Границы кода языка ассемблера задаются двумя специальными командами #asm и #endasm. Все, что находится между ними, прямо пересылается в выходной файл компилятора. Подстановки в макроопределениях при этом не выполняются. Такие последовательности команд разрешены везде, где синтаксис допускает использование описаний или операторов. Чтобы с успехом пользоваться указанными возможностями, необходимо, конечно, знать о том, как компилятор использует регистры ЦП, как вызываются функции и как работают операционная система и аппаратура. Генерация кода компилятором Смолл-Си описана в гл.18.

[1] Под переносимостью здесь понимается возможность переноса программы с одной ЭВМ на другую или из одной операционной системы в другую. - *Прим. перев.*

ЧАСТЬ 3

КОМПИЛЯТОР СМОЛЛ-СИ

В ч.2 вопросы организации ввода-вывода, а также управления программой и памятью не рассматривались, так как спецификация языка Си не включает в себя эти операции. Цель такого подхода - осуществление этих операций для каждой реализации языка с помощью стандартных функций. Стандартные функции можно разработать таким образом, что интерфейс с различными операционными системами будет невидим программисту и, следовательно, для программиста они будут выглядеть одинаково, независимо от того, какую операционную систему или ЭВМ он использует. И действительно, для большинства применений разработан достаточно стандартный набор функций.

В этой части описываются стандартные функции Смолл-Си, а также рассматриваются особенности практического использования компилятора: переадресация ввода-вывода, аргументы командной строки, вызов компилятора, генерация кода, эффективность программ и компиляция компилятора.

После гл.17, в которой описываются стандартные функции, вы будете иметь всю информацию, необходимую для того, чтобы успешно писать и компилировать программы на языке Смолл-Си. Назначение остальных глав - помочь в изучении компилятора, написании более эффективных программ и использовании компилятора для создания его новых версий.

ГЛАВА 16

ИТЕРФЕЙС С ПОЛЬЗОВАТЕЛЕМ

В версии 2.1 компилятора Смолл-Си использованы две идеи, заимствованные из операционной системы (ОС) UNIX: метод переадресации входного и выходного файлов во время выполнения программы и метод передачи аргументов программе. Эти концепции существенны для пользователя компилятора и других программ на языке Смолл-Си. Не в каждой реализации компилятора Смолл-Си эти концепции используются, однако в большинстве случаев они нашли свое отражение.

ПЕРЕАДРЕСАЦИЯ ВВОДА-ВЫВОДА

В программах на языке Смолл-Си файлы идентифицируются целыми константами, называемыми *описателями файлов* (fd - file descriptor). Как показано в табл.16.1, значения некоторых из этих описателей заранее заданы.

Таблица 16.1. **Стандартные назначения описателя файла**

fd	Имя	Назначение по умолчанию	Режим открытия	Комментарий
0	stdin	Консоль	Чтение	Переадресуемый
1	stdout	Консоль	Запись	Переадресуемый
2	stderr	Консоль	Запись	Не переадресуемый

Идея стандартных назначений описателей файлов состоит в том, что программа начинает работать при трех уже открытых стандартных файлах: *стандартном файле ввода* (stdin), *стандартном файле вывода* (stdout) и *стандартном файле ошибок* (stderr). По умолчанию все они назначаются на консоль (stdin - на клавиатуру, stdout и stderr - на экран). Если не было других назначений при вызове программы, то весь ввод данных из файла stdin происходит с клавиатуры консоли. О конце файла пользователь сообщает, вводя управляющий символ, зависящий от конкретной реализации (обычно управляющий символ D или Z). Аналогично вывод в файлы stdout и stderr по умолчанию осуществляется на экран.

Задав в командной строке, с помощью которой вызывается программа, спецификацию переадресации, пользователь может для данного прогона программы изменить назначения, заданные по умолчанию. Из трех стандартных файлов таким образом можно переадресовать только два: stdin и stdout. Предполагается, что файл stderr должен использоваться для сообщений об ошибках и другой подобной информации, которая всегда выводится на экран, независимо от того, был или нет переадресован файл stdout.

Спецификации переадресации могут быть помещены в любом месте командной строки после имени программы. Для того чтобы переадресовать файл stdin, непосредственно перед именем существующего файла вводится символ <. При этом весь ввод будет производиться не с клавиатуры, а из заданного файла. Формат имени файла следует из соглашений, принятых в конкретной операционной системе.

Символ >, за которым сразу следует имя файла, переадресует файл stdout. При этом весь вывод в файл stdout будет идти не на экран консоли, а в заданный файл. Если файл отсутствует, то

он автоматически создается. Если же файл существует, то запись осуществляется непосредственно в этот файл.

В ОС UNIX устройствам ввода-вывода присвоены специальные имена, которые могут использоваться в спецификации переадресации подобно именам обычных файлов. В версиях компилятора Смолл-Си для ОС CP/M тем же самым целям служат имена логических устройств, например LST:. В других операционных системах для этого могут использоваться иные средства. Файлы stdin и stdout могут быть переадресованы одновременно, и, кроме того, спецификации переадресации и аргументы в командной строке могут быть перемешаны в любом порядке. Порядок спецификаций переадресации может быть любым.

Простая программа копирования может быть вызвана, например, с помощью команд, указанных в табл.16.2.

Таблица 16.2. **Переадресация стандартных файлов ввода и вывода**

Команда	Комментарий
`copy <abc`	Скопировать файл abc на консоль
`copy <abc >def`	Скопировать файл abc в файл def
`copy >abc`	Скопировать ввод с клавиатуры в файл abc

ПАРАМЕТРЫ КОМАНДНОЙ СТРОКИ

Из программы можно получить доступ к параметрам командной строки (любым печатаемым символам, кроме спецификаций переадресации, в строке, следующей за именем программы). Чтобы сделать это, нужно описать функцию main следующим образом:

```
main (argc, argv) int argc, *argv;
```

Однако, если в программе не используются параметры командной строки, то можно написать

```
main ()
```

Команда

```
prog x zz<abc 1234
```

при вызове программы установит стек так, как показано на рис.16.1. Строка <abc не передается, так как она является спецификацией переадресации. Все остальные параметры выделяются в отдельные строки символов, и каждая такая строка оканчивается нулевым байтом. Устанавливается массив указателей на эти строки, для которого переменная argc определяет число элементов, а переменная argv - адрес первого элемента.

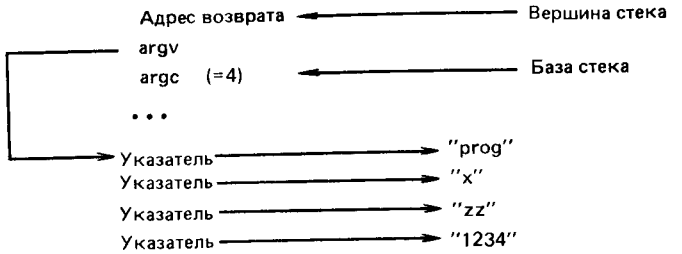

Рис.16.1. Аргументы, передаваемые программам Смолл-Си

Подобно любым другим переменным, argc и argv могут модифицироваться программой. Для получения с помощью переменных argc и argv последовательности параметров в соответствии с их положением в командной строке разработана функция getarg, входящая в библиотеку Смолл-Си (см. гл.17).

Назначение параметров командной строки может быть любым, но чаще всего это или имена файлов, или *ключи* (параметры, предназначенные для управления выбором, конкретных опций[1]). Обычно ключ идентифицируется дефисом, стоящим перед ним.

ВЫЗОВ КОМПИЛЯТОРА

Чтобы сообщить компилятору, откуда получить входной файл, куда послать выходной и какие опции использовать во время работы, служат описанные выше спецификации переадресации ввода-вывода и параметры, передаваемые в командной строке.

По умолчанию компилятор Смолл-Си получает входную информацию из стандартного файла ввода, который может быть переадресован с клавиатуры консоли на другое устройство или файл на диске. Однако если в командной строке появляется одно или несколько имен файлов (не спецификаций переадресации), то вместо стандартного ввода компилятор читает названные файлы в перечисленном порядке. Если для имен этих файлов расширения не заданы, то при работе под управлением ОС CP/M по умолчанию задается расширение C.

По умолчанию компилятор осуществляет вывод в стандартный файл вывода. Вывод может быть переадресован с экрана консоли на другое устройство или в файл на диске. Если для ввода заданы имена файлов, а файл stdout не переадресован в файл на диске, то в качестве выходного компилятор использует файл на диске с тем же именем, что и у входного файла, но с расширением MAC - расширением, принятым по умолчанию для исходных файлов макроассемблера MACRO-80.

[1] Опции - это дополнительные возможности, предназначенные для модификации основного режима работы программы. - *Прим.перев.*

Эти соглашения по умолчанию легко изменить, модифицировав функцию openin в файле CC11.C и затем откомпилировав заново компилятор (см. гл.20). Заметим, что спецификации переадресации не имеют расширений по умолчанию.

Компилятор особенно легко изучать, используя файлы stdin и stdout. Вызвав компилятор и вводя с клавиатуры тестовую программу, можно видеть на экране, какие команды на языке ассемблера генерируются на выходе.

Работа компилятора может быть прервана одним из двух способов. Он делает паузу, если введен управляющий символ Ctrl-S, и продолжает работу при вводе любого другого символа. При вводе управляющего символа Ctrl-C работа компилятора аварийно прекращается.

Работой компилятора управляют следующие ключи:

1. Ключ -m позволяет *контролировать* работу компилятора, так как он будет выдавать в стандартный файл ошибок каждую строку с описанием функции. Этот ключ помогает определять места ошибок по именам функций, содержащих эти ошибки.

2. Ключ -a обеспечивает при выводе сообщения об ошибке выдачу в стандартный файл ошибок символа *звукового сигнала* (управляющего символа Ctrl-G).

3. По ключу -p компилятор после выдачи каждого сообщения об ошибке делает *паузу*. Работа продолжается после ввода с клавиатуры символа возврата каретки.

4. По ключу -l#, где # - описатель файла, компилятор Смолл-Си выдает в заданный файл листинг исходной программы. Если символ # имеет значение 1 (stdout), то листинг будет идти вперемешку с выходным файлом. В этом случае перед каждой строкой исходной программы ставится точка с запятой, что превращает эту строку в комментарий для ассемблера. Если символ # имеет значение 2 (stderr), то листинг выдается на консоль. Если этот ключ не задан, то листинг не выдается.

5. Машинно-независимая оптимизация генерируемого кода производится всегда. По ключу -o происходит дальнейшее уменьшение размера программы, даже за счет скорости ее выполнения.

6. Ключ -b# задается только тогда, когда при компиляции компилятора не было предусмотрено его использование совместно с динамическим загрузчиком. В этом случае сегменты программы должны объединяться во время ассемблирования, а не во время загрузки. По ключу -b# нумерация меток начинается со значения символа #. Если символ # имеет значение 0 (значение по умолчанию), то происходит компиляция всей программы. В этом случае к программе добавляются зависящие от реализации начальный и конечный коды, предназначенные для связи программы с окружающей программной средой. Если символ # имеет значение 1, то компилируется первая часть программы, состоящей из нескольких частей. В этом случае будет добавлен только на-

чальный код. Если символ # имеет значение между 1 и 9000, то это указывает на промежуточную часть программы. В этом случае ни начальный, ни конечный коды не добавляются. И наконец: если символ # имеет значение 9000, то компилируется последняя часть программы, и поэтому будет добавлен только конечный код. Значения символа # должны выбираться таким образом, чтобы избежать конфликтов с метками в других частях программы.

7. Нулевой ключ (или любой неопределенный ключ) приводит к выдаче на экран консоли строки

usage: cc [file]...[-m] [-a] [-p] [-l#] [-o] [-b#]

и завершению работы компилятора. Этой строки может быть достаточно, чтобы напомнить правильный формат вызова компилятора.

В табл.16.3 приведены примеры команд вызова компилятора и описано действие каждой из них.

Таблица 16.3. **Вызов компилятора**

Команда	Комментарий
`cc`	Компилировать файл, вводимый с консоли, с выводом на консоль
`cc <prog.c -l1 -p`	Компилировать файл prog.c с выводом на консоль, выдать исходный текст в виде комментариев к выводу, при ошибках делать паузу
`cc <abc.c >xyz.mac -m`	Компилировать файл abc.c в выходной файл xyz.mac, контролировать процесс, выдавая на консоль строки с заголовками функций
`cc abc def -o -a`	Компилировать файл abc.c, а затем файл def.c в единую программу (файл abc.mac), оптимизировать объем за счет скорости, при ошибках выдавать звуковой сигнал

ГЛАВА 17

СТАНДАРТНЫЕ ФУНКЦИИ

В этой главе представлен набор функций для выполнения операций ввода-вывода, преобразований формата данных, манипуляций со строками и выполнения других задач. Большинство из этих функций имеет своих двойников в библиотеках, доступных пользователям ОС UNIX. Однако некоторые из них отсутствуют в

этих библиотеках. Ниже такие функции обозначаются как функции Смолл-Си. Использование функций Смолл-Си будет влиять на возможность компиляции программ другими компиляторами Си, если только сами функции не переносятся вместе с программами. Часть из описанных здесь функций отсутствует в некоторых реализациях компилятора Смолл-Си. Чтобы определить, какие функции поддерживаются, следует ознакомиться с документацией, поставляемой вместе с вашим дистрибутивным диском.

Для описателей файлов и для значений, возвращаемых этими функциями, используются некоторые символы, определенные в файле STDIO.H, который должен включаться в каждую программу. Ниже приведен список этих символов:

```
#define stdin  0   /* fd стандартного файла ввода */
#define stdout 1   /* fd стандартного файла вывода */
#define stderr 2   /* fd стандартного файла ошибок */
#define ERR   -2   /* возвращаемое значение условия ошибки */
#define EOF   -1   /* значение, возвращаемое при обнаружении конца файла */
#define NULL   0   /* значение нулевого символа */
```

В различных реализациях некоторые значения из этого списка могут отличаться, однако, если вместо самих констант в программах используются эти символические имена, совместимость между различными реализациями будет обеспечена.

ФУНКЦИИ ВВОДА-ВЫВОДА

В функциях *стандартной библиотеки* ввода-вывода ОС UNIX в качестве средства идентификации конкретных файлов принят указатель на находящуюся в памяти файловую структуру. Однако в ОС UNIX функции нижнего уровня вместо этого используют целую константу, называемую описателем файла (fd). В Смолл-Си применяется последняя схема, хотя многие функции Смолл-Си аналогичны стандартным функциям ОС UNIX. Тем не менее из-за этого различия проблем с совместимостью не возникает, так как для функций ввода-вывода не имеет значения, является переменная, используемая в качестве определителя файла, указателем или небольшим целым числом. Трудности могут возникнуть только при компиляции с помощью компилятора Смолл-Си программ, написанных для полной версии языка Си, если для получения доступа к структуре управления файлом в этих программах используется указатель. Но, так как в языке Смолл-Си реализовано подмножество полной версии языка Си, программы будут в большинстве случаев переноситься другим путем.

fopen (name, mode) char *name, *mode;

Эта функция пытается открыть файл, заданный строкой символов (заканчивающейся нулевым байтом), на которую указывает параметр name. Параметр mode указывает на строку, определяю-

щую режим использования открываемого файла, и может принимать следующие значения:

"r" - чтение,
"w" - запись,
"a" - добавление.

В режиме *чтение* существующий файл открывается для ввода. В режиме *запись* создается новый файл или открывается старый, который усекается таким образом, что запись начинается с начала файла. В режиме *добавление* можно осуществлять запись в конец существующего файла или в начало нового. В дополнение к этому возможны следующие режимы обновления файла:

"r+" - чтение с обновлением,
"w+" - запись с обновлением,
"a+" - добавление с обновлением.

При открытии файлов эти режимы идентичны соответствующим режимам без обновления файлов, однако они позволяют переходить с ввода на вывод и наоборот, перемежая вызов функций ввода и вывода. Следующая операция *чтения* или *записи* начинается с той точки, где кончилась предыдущая, если только программа не выполняла операций поиска или перемотки. Если попытка открыть файл прошла успешно, то функция fopen возвращает значение fd для открытого файла; иначе она возвращает символ NULL. Затем это значение fd используется в последующих вызовах функций ввода-вывода для идентификации данного файла. Без предварительного вызова функции fopen можно обращаться только к стандартным файлам (см. табл.16.1).

freopen (name,mode, fd) char *name, *mode; int fd;

Эта функция закрывает открытый ранее файл, определяемый fd, и открывает новый файл, имя которого содержится в символьной строке (заканчивающейся нулевым байтом), на которую указывает параметр name. Параметр mode указывает на символьную строку, задающую режим открытия файла (точно так же, как и для функции fopen). Функция возвращает новое значение fd, если открытие прошло успешно, и символ NULL, если произошла ошибка при закрытии старого файла или при открытии нового. Заметим, однако, что так как значение fd для стандартного файла ввода равно нулю, то в этом случае невозможно определить, была ли ошибка.

fclose (fd) int fd;

Эта функция закрывает файл, задаваемый fd. Если в буфере файла находились какие-либо новые данные, то сначала они записываются на диск. Функция возвращает символ NULL, если не было ошибки, и ненулевое значение в противном случае.

getc (fd) int fd;
fgetc (fd) int fd;

Эти функции возвращают следующий символ файла, на который указывает fd. Если в файле больше нет символов или произошла ошибка, они возвращают символ EOF. Конец файла обнаруживается в том случае, если встречен стандартный для данной реализации символ конца файла или физический конец файла.

ungetc (c, fd) char c; int fd;

Эта функция логически (не физически) записывает символ обратно в файл, заданный fd. При следующем чтении из данного файла этот символ будет получен первым. При каждом обращении может быть записан только один символ. Функция возвращает записанный символ при успешном выполнении и символ EOF, если еще хранится записанный ранее символ или же символ с имеет значение EOF. Записать символ EOF в файл нельзя. Выполнение операции поиска или перехода к началу файла вызывает потерю записанного символа.

getchar ()

Эта функция эквивалентна функции fgetc (stdin).

fgets (str, sz, fd) char *str; int sz, fd;

Эта функция пересылает из файла, заданного fd, в память до sz-1 символов, начиная с адреса, заданного параметром str. Ввод заканчивается после передачи признака начала новой строки. Перед признаком новой строки или в последнюю позицию данной строки, если признак не был обнаружен, записывается нулевой символ. Функция fgets возвращает значение str при успешном выполнении или же символ NULL, если обнаружен конец файла или была ошибка.

fread (ptr, sz, cnt, fd) char *ptr; int sz, cnt, fd;

Эта функция пересылает из файла, заданного fd, в память cnt элементов данных длиной sz байтов каждый, начиная с адреса, заданного параметром ptr. Вызывающей программе возвращается число действительно прочитанных элементов. Если был обнаружен конец файла, то это число может быть меньше cnt. Функция fread выполняет *двоичную* пересылку: она не преобразует последовательность "возврат-каретки/перевод-строки" в признак начала новой строки и не обращает внимания на признаки конца файла. Она распознает только физический конец файла. Чтобы определить, что данные кончились, следует вызвать функцию feof, а для обнаружения ошибок - функцию ferror.

read (fd, ptr, cnt) int fd, cnt; char *ptr;

Эта функция пересылает из файла, заданного fd, в память cnt байтов данных, начиная с адреса, заданного параметром ptr. Вы-

зывающей программе возвращается число действительно прочитанных байтов. Если был обнаружен конец файла, то это число может быть меньше cnt. Функция read выполняет *двоичную* пересылку: она не преобразует последовательность "возврат-каретки/перевод-строки" в признак начала новой строки и не обращает внимания на признаки конца файла. Она распознает только физический конец файла. Чтобы определить, что данные кончились, следует вызвать функцию feof, а для обнаружения ошибок - функцию ferror.

gets (str) char *str;

Эта функция пересылает символы из файла stdin в память, начиная с адреса, заданного параметром str. Ввод заканчивается при обнаружении признака начала новой строки, но сам этот признак не пересылается. Введенная строка заканчивается нулевым символом. При успешном выполнении функция gets возвращает значение str, а при обнаружении конца файла или ошибки - символ NULL. Так как эта функция может пересылать любой объем данных, то необходимо контролировать длину вводимой строки, чтобы удостовериться, что она не выходит за пределы отведенной области.

feof (fd) int fd;

Эта функция возвращает ненулевое значение, если достигнут конец файла, заданного fd; иначе она возвращает символ NULL.

ferror (fd) int fd;

Эта функция возвращает ненулевое значение, если в любой момент после открытия файла, заданного fd, была обнаружена ошибка; иначе она возвращает символ NULL.

clearerr (fd) int fd;

Эта функция сбрасывает признак ошибки для файла, заданного fd.

putc (c, fd) char c; int fd;
fputc (c, fd) char c; int fd;

Эти функции записывают символ c в файл, заданный fd. При успешном выполнении они возвращают сам символ, иначе - символ EOF. Если c является признаком начала новой строки, то записывается пара символов "возврат-каретки" и "перевод-строки".

putchar (c) char c;

Эта функция эквивалентна функции fputc (c, stdout).

fputs (str, fd) char *str; int fd;

Эта функция записывает символы, начиная с адреса, заданного параметром str, в файл, заданный fd. Символы записываются по-

следовательно до тех пор, пока не будет обнаружен нулевой байт. Нулевой байт не записывается, и признак начала новой строки не добавляется.

puts (str) char *str;

Эта функция действует подобно функции fputs (str, stdout), но, кроме того, добавляет при выводе признак начала новой строки.

fwrite (ptr, sz, cnt, fd) char *ptr; int sz, cnt, fd;

Эта функция записывает в файл, заданный fd, из памяти cnt элементов данных длиной sz байтов, начиная с адреса, на который указывает ptr. Она возвращает число записанных элементов. При ошибке число записанных элементов может быть меньше cnt. В любом случае для обнаружения ошибки следует вызвать функцию ferror. Функция fwrite выполняет *двоичную* пересылку и не преобразует признак начала новой строки в последовательность 'возврат-каретки/перевод-строки".

write (fd, ptr, cnt) int fd, cnt; char *ptr;

Эта функция записывает в файл, заданный fd, cnt байтов из памяти, начиная с адреса, заданный параметром ptr. Она возвращает число записанных байтов. При ошибке число записанных байтов может быть меньше cnt. В любом случае для обнаружения ошибки следует вызвать функцию ferror. Функция write выполняет *двоичную* пересылку и не преобразует признак начала новой строки в последовательность "возврат-каретки/перевод-строки".

fflush (fd) int fd;

Эта функция осуществляет пересылку данных из системного буфера в файл. Обычно данные, записываемые в файл на диске, содержатся в буфере, расположенном в памяти, до тех пор, пока: 1) не заполнится буфер, 2) область буфера не потребуется для хранения другого находящегося на диске блока данных или 3) не будет закрыт файл. При успешном выполнении функция fflush возвращает символ NULL, а при ошибке - символ EOF. Эта функция вызывается функцией fclose.

cseek (fd, offset, from) int fd, offset, from;

Эта функция Смолл-Си позиционирует указатель текущего байта в файле, заданном fd, на начало 128-байтовой записи, отстоящей на offset позиций от первой записи, текущей записи или конца файла, в зависимости от значения параметра from (0, 1 или 2 соответственно). Последующие запись или чтение продолжаются с этой точки. Функция возвращает символ NULL при успешном выполнении и символ EOF в противном случае.

rewind (fd) int fd;

Эта функция Смолл-Си позиционирует указатель текущего байта в файле, заданном fd, на начало файла. Она эквивалентна поиску первого байта файла. Функция возвращает символ NULL при успешном выполнении и символ EOF в противном случае.

ctell (fd) int fd;

Эта функция Смолл-Си возвращает позицию текущей записи файла, заданного fd. Возвращаемое значение является смещением текущей 128-байтовой записи относительно первой записи файла. Если значение fd соответствует не файлу на диске, то возвращается значение -1.

delete (name) char *name;

unlink (name) char *name;

Эти функции удаляют файл. Имя файла задано в строке, на которую указывает параметр name и которая заканчивается нулевым символом. Функции возвращают символ NULL при успешном выполнении и символ EOF в противном случае.

iscons (fd) int fd;

Эта функция возвращает ненулевое значение, если значение fd соответствует консоли; иначе она возвращает символ NULL.

isatty (fd) int fd;

Эта функция возвращает ненулевое значение, если значение fd соответствует не файлу на диске; иначе она возвращает символ NULL.

ФУНКЦИИ ФОРМАТИРОВАННОГО ВВОДА-ВЫВОДА

printf (str, arg1, arg2, ...) char *str;

Эта функция записывает в стандартный файл вывода форматированную символьную строку, заданную параметром str и содержащую массив символов, который заканчивается нулевым символом; отдельные группы символов этого массива соответствуют аргументам функции

arg1, arg2, ...

Функция возвращает общее число записанных символов. Строка str называется *управляющей*. Эта строка должна быть обязательно задана, остальные же аргументы являются необязательными. Управляющая строка содержит обычные символы и группы символов, называемые *спецификациями преобразования*. Каждая спецификация преобразования информирует функцию printf о том, как преобразовать соответствующий аргумент в символьную строку

для вывода. При выводе преобразованный аргумент заменяет спецификацию преобразования. Спецификация преобразования начинается с символа %, а заканчивается одной из букв b, c, d, o, s, u или x.

Между началом и концом спецификации преобразования в указанном порядке и без промежуточных пробелов могут находиться: знак минус (-), целая десятичная константа (nnn) и десятичная дробь (.mmm). Все эти элементы являются необязательными. Часто в спецификациях преобразования они отсутствуют. Знак минус показывает, что строка, получаемая в результате заданного преобразования аргумента, будет при выводе сдвинута к левому краю поля этого аргумента. Целая десятичная константа задает минимальную ширину данного поля (в символах). Если оно мало, то будет расширено до необходимых размеров, но в любом случае будет выведено, по меньшей мере, заданное число позиций. Десятичная дробь применяется в том случае, когда аргумент сам является символьной строкой (точнее, адресом символьной строки). При этом десятичная дробь указывает максимальное число символов, которые должны быть взяты из строки. Если в спецификации отсутствует десятичная дробь, то обрабатывается вся строка.

Буква, заканчивающая спецификацию, указывает на тип преобразования аргумента. Этим буквам соответствуют следующие типы преобразования:

- b Аргумент рассматривается как целое число без знака и преобразуется для вывода в *двоичном* формате. Ведущие нули не генерируются. При использовании этой спецификации следует иметь в виду, что она существует только в Смолл-Си.
- c Аргумент выводится как *символ* без какого-либо преобразования. В этом случае старший байт игнорируется.
- d Аргумент рассматривается как целое число со знаком и преобразуется для вывода в строку *десятичных* цифр (возможно, со знаком). Знак является самым левым символом; это может быть пробел для положительных чисел или знак минус для отрицательных.
- o Аргумент рассматривается как целое число без знака и преобразуется для вывода в *восьмеричном* формате. Ведущие нули не генерируются.
- s Аргумент является адресом символьной *строки*, заканчивающейся нулевым символом. Строка выводится без преобразования, однако при этом в спецификации задаются минимальная ширина и максимальный размер поля.
- u Аргумент рассматривается как беззнаковое целое число. Он преобразуется в десятичное число *без знака*. Ведущие нули не генерируются.

x Аргумент рассматривается как беззнаковое целое число. Он преобразуется для вывода в *шестнадцатеричном* формате. Ведущие нули не генерируются.

Если за символом % следует не правильная спецификация, а что-либо другое, то сам он игнорируется, а следующий символ выводится без изменения. Таким образом, %% выведется как %.

Функция printf просматривает строку слева направо, пересылая символы в файл stdout до тех пор, пока не найдет символ %. Тогда она выделяет первую спецификацию преобразования и применяет ее для первого аргумента (следующего за управляющей строкой). Результат записывается в файл stdout. После этого функция printf возобновляет запись данных из управляющей строки, пока не найдет еще одну спецификацию преобразования, которая применяется, в свою очередь, ко второму аргументу. Эта процедура продолжается до тех пор, пока не будет исчерпана управляющая строка. Результатом является форматированная строка, содержащая как обычные символы из управляющей строки, так и переменные. В табл.17.1 приведены примеры, которые могут помочь пониманию правил использования функции printf. Так как данная функция проверяет число аргументов, то в вызывающей ее программе не должен быть определено имя NOCCARGC (см. гл.12).

Таблица 17.1. Примеры использования функции printf

Управляющая строка	Аргументы	Вывод
"oct %o, dec %d"	127, 127	oct 177, dec 127
"%u=%x"	- 1, - 1	65535 = FFFF
"%d%% interest"	10	10% interest
"(%6d)"	55	(55)
"(%-6d)"	123	(123)
"The letter is %c."	'A'	The letter is A
"Call me %s."	"Fred"	Call me Fred.
"Call me %.3s."	"Fred"	Call me Fre.
"Call me %4.3s."	"Fred"	Call me Fre.

frpintf (fd, str, apg1, arg2, ...) int fd; shar *str;

Эта функция подобна функции printf, однако вывод производится в файл, заданный fd. Так как данная функция проверяет число аргументов, то в вызывающей ее программе не должно быть определено имя NOCCARGC (см. гл.12).

scanf (str, arg1, arg2, ...)char *str;

Эта функция читает поля из стандартного файла ввода, преобразует их содержимое во внутренний формат в соответствии со

спецификациями преобразования, содержащимися в управляющей строке str, и записывает в места, заданные аргументами

arg1, arg2, ...

Функция возвращает число прочитанных полей. Поле на входе - это строка идущих подряд графических символов. Она заканчивается одним из разделителей (пробел, символ табуляции или признак перехода на новую строку) или же, если задана максимальная ширина поля, - концом этого поля. Поле обычно начинается с первого графического символа после предыдущего поля; таким образом, предшествующие ему разделители пропускаются. Так как при поиске следующего поля признак перехода на новую строку пропускается, функция scanf читает столько входных строк, сколько требуется для того, чтобы исчерпать все спецификации преобразования, заданные в ее управляющей строке. Каждый аргумент, следующий за управляющей строкой, задает определенный адрес.

Управляющая строка содержит спецификации преобразования и разделители (которые игнорируются). Каждая спецификация преобразования указывает функции scanf, как преобразовать во внутренний формат соответствующее поле, а каждый аргумент, следующий за str, задает адрес, по которому должно быть записано соответствующее преобразованное поле. Символ % задает начало спецификации преобразования, а одна из букв b, c, d, o, s, u или x - ее конец.

Между началом и концом спецификации преобразования могут находиться звездочка и/или десятичная целая константа без разделяющих их пробелов. Оба эти подполя необязательны. В действительности часто можно увидеть спецификации преобразования без этих подполей. Звездочка указывает на то, что соответствующее поле на входе должно быть пропущено. Спецификации пропуска полей не должны иметь соответствующих им аргументов. Число задает максимальную ширину поля в символах. Если оно присутствует, то поле заканчивается после просмотра заданного числа символов, даже если разделитель при этом не был найден. Однако, если до конца поля был обнаружен разделитель, поле заканчивается в этой точке.

Буква в конце спецификации преобразования задает тип преобразования, применяемого к данному полю. Этим буквам соответствуют следующие типы преобразования:

b Поле обрабатывается как *двоичное* целое число и преобразуется в целое значение. Соответствующий аргумент должен быть адресом целого числа. Ведущие нули игнорируются. При использовании этой спецификации имейте в виду, что она существует только в языке Смолл-Си.

c Поле обрабатывается как одиночный *символ* без какого-либо преобразования. Эта спецификация запрещает пропуск разделителя. Аргументом для такого поля должен быть адрес символа.

d Поле обрабатывается как *десятичное* целое число (возможно, со знаком) и преобразуется в целое значение. Соответствующий аргумент должен быть адресом целого. Ведущие нули игнорируются.

o Поле обрабатывается как *восьмеричное* целое число и преобразуется в целое значение. Соответствующий аргумент должен быть адресом целого. Ведущие нули игнорируются.

s Поле обрабатывается как символьная *строка* и записывается по адресу символа, заданному его аргументом. Строка при записи заканчивается нулевым символом. По этому адресу должно быть достаточно места, чтобы записать строку и нулевой символ. Напомним, что для предотвращения переполнения можно задать максимальную ширину поля. Если задана спецификация %s, то будет прочитан следующий графический символ, а для спецификации %c - следующий символ независимо от того, что он собой представляет.

u Поле обрабатывается как *беззнаковое* целое десятичное число и преобразуется в целое значение. Соответствующий аргумент должен быть адресом целого. Ведущие нули игнорируются. При использовании этой спецификации имейте в виду, что она сущесвует только в языке Смолл-Си.

x Поле обрабатывается как *шестнадцатеричное* число и преобразуется в целое значение. Соответствующий аргумент должен быть адресом целого. Ведущие нули или ведущие последовательности 0x и 0X игнорируютсяся.

Функция scanf просматривает управляющую строку слева направо, обрабатывая поля до тех пор, пока не кончится управляющая строка или не будет найдено поле, не соответствующее его спецификации преобразования. Если значение, возвращенное функцией scanf, меньше числа спецификаций проебразования, то это значит, что была обнаружена ошибка или же был достигнут конец входного файла. Если не было обработано ни одного поля, то возвращается символ EOF.

Если при выполнении оператора

```
scanf("%s  %c  %c  %*s  %d  %3d  %d",
      str, &c1, &c2       &i1, &i2, &i3);
```

на входе задано

```
"abc defg  -12  345678 9"
```

то будут выведены следующие значения:

```
"abc\0" в str
    ' ' в c1
    'd' в c2
    -12 в i1
    345 в i2
    678 в i3
```

Следующий ввод из файла начнется с пробела, стоящего после цифр 345678. Так как эта функция проверяет число аргументов, то в вызывающей программе не должно быть определено имя NOCCARGC (см. гл.12).

fscanf (fd, str, arg1, arg2, ...) int fd; char *str;

Эта функция действует подобно функции scanf, но при этом ввод идет из файла, заданного fd. Так как эта функция проверяет число аргументов, то в вызывающей программе не должно быть определено имя NOCCARGC (см. гл.12).

ФУНКЦИИ ФОРМАТНЫХ ПРЕОБРАЗОВАНИЙ

atoi (str) char *str;

Эта функция преобразует десятичное число, представленное строкой, на которую указывает str, в целое и возвращает его значение. Ведущие разделители пропускаются; самой левой цифре может предшествовать знак (+ или -). Преобразование заканчивается на первом нецифровом символе.

atoib (str, base) char *str; int base;

Эта функция Смолл-Си преобразует беззнаковое целое число с базой base, представленное строкой str, в целое и возвращает его значение. Ведущие разделители пропускаются. Преобразование заканчивается на первом нецифровом символе.

itoa (nbr, str) int nbr; char *str;

Эта функция представляет число nbr в виде десятичной символьной строки str. Результат в этой строке сдвигается влево до границы строки; если число nbr отрицательное, то перед ним ставится знак минус. Строка заканчивается нулевым символом, и ее длина должна быть достаточной для того, чтобы в этой строке поместился результат.

itoab (nbr, str, base) int nbr; char *str; int base;

Эта функция Смолл-Си преобразует беззнаковое целое число nbr в его представление в символьной строке str с базой base. Результат в строке str сдвигается влево до границы строки. Строка заканчивается нулевым символом, и ее длина должна быть достаточной для того, чтобы в этой строке поместился результат.

dtoi (str, nbr) char *str; int *nbr;

Эта функция Смолл-Си преобразует десятичное число, которое может содержать знак и находится в символьной строке str, в целое число nbr и возвращает длину найденного числового поля. Преобразование заканчивается, когда функция находит конец строки или нецифровой символ. Функция dtoi обрабатывает 16-разрядные числа: знак числа и не более пяти цифр. Если абсолютное значение числа превышает 32767, функция возвращает символ ERR.

otoi (str, nbr) char *str; int *nbr;

Эта функция Смолл-Си преобразует беззнаковое восьмеричное число, находящееся в строке str, в целое число nbr и возвращает длину найденного поля восьмеричных цифр. Преобразование заканчивается, когда в строке str обнаруживается невосьмеричная цифра. Функция otoi обрабатывает 16-разрядные числа: до шести цифр. Числа, значения которых превышают восьмеричное значение 177777, приводят к возврату функцией символа ERR.

utoi (str, nbr) char *str; int *nbr;

Эта функция Смолл-Си преобразует беззнаковое десятичное число, представленное символьной строкой str, в целое число nbr и возвращает длину найденного числового поля. Преобразование заканчивается, когда функция обнаруживает конец строки или символ, не являющийся десятичной цифрой. Функция utoi обрабатывает 16-разрядные целые числа: до пяти цифр. Число, значение которого превышает значение 65535, вызывает возврат функцией символа ERR.

xtoi (str, nbr) char *str; int *nbr;

Эта функция Смолл-Си преобразует шестнадцатеричное число в символьной строке str в целое число nbr и возвращает длину найденного шестнадцатеричного поля. Преобразование заканчивается, когла в строке str обнаруживается символ, не являющийся шестнадцатеричной цифрой. Функция xtoi обрабатывает 16-разрядные целые числа: до четырех цифр. При большем числе цифр возвращается символ ERR.

itod (nbr, str, sz) int nbr,sz; char *str;

Эта функция Смолл-Си преобразует число nbr в символьную строку str; если это число отрицательное, то строка содержит знак минус. Результат в строке сдвигается вправо и дополняется слева пробелами. Если строка назначения слишком мала, то знак и старшие разряды будут опущены. Функция itod возвращает строку str. Значение sz равно длине строки. Если sz больше нуля, то в позицию str[sz-1] помещается нулевой байт. Если sz равно нулю, то длина строки определяется по первому нулевому

байту, следующему за строкой str. Если sz меньше нуля, то используются все sz символов строки, включая последний.

itoo (nbr, str, sz) int nbr, sz; char *str;

Эта функция Смолл-Си преобразует число nbr в восьмеричную символьную строку str. Результат в строке сдвигается вправо и дополняется пробелами. Если строка назначения слишком мала, то старшие разряды будут опущены. Функция itoo возвращает строку str. Значение sz равно длине строки. Если sz больше нуля, то в позицию str[sz-1] помещается нулевой байт. Если sz равно нулю, то длина строки определяется по первому нулевому байту, следующему за строкой str. Если sz меньше нуля, то используются все sz символов строки, включая последний.

itou (nbr, str, sz) int nbr, sz; char *str;

Эта функция Смолл-Си преобразует число nbr в десятичное число без знака в символьной строке str. Она действует подобно функции itod, но при этом старший разряд входит в значение числа nbr.

itox (nbr, str, sz) int nbr, sz; char *str;

Эта функция Смолл-Си преобразует число nbr в шестнадцатеричную символьную строку str. Результат в строке str сдвигается вправо и дополняется слева пробелами. Если строка назначения слишком мала, то старшие значащие разряды будут опущены. Функция itox возвращает строку str. Если sz больше нуля, то в позицию str[sz-1] помещается нулевой байт. Если sz равно нулю, то длина строки определяется по первому нулевому байту, следующему за строкой str. Если sz меньше нуля, то используются все sz символов строки, включая последний.

ФУНКЦИИ ОБРАБОТКИ СТРОК

left (str) char *str;

Эта функция Смолл-Си сдвигает символьную строку str влево до границы строки. Строка, начиная с первого символа, не являющегося пробелом, до нулевого байта сдвигается по адресу str.

strcat (dest, sour) char *dest, *sour;

Эта функция добавляет строку sour в конец строки dest, возвращая строку dest. Нулевой символ в конце строки dest замещается ведущим символом строки sour. Новая строка dest заканчивается нулевым символом. Область, зарезервированная для строки dest, должна быть достаточной для того, чтобы в ней поместился результат.

strncat (dest, sour, n) char *dest, *sour; int n;

Эта функция действует подобно функции strcat, но при этом из исходной строки в строку назначения передается не более n символов.

strcmp (str1, str2) char *str1, *str2;

Эта функция возвращает целое число, меньшее, равное или большее нуля, в зависимости от того, меньше, равна или больше (по числовому значению) строка str1 строки str2. Функция производит посимвольное сравнение строк от начала до конца, пока не будут обнаружены несовпадающие символы. Сравнение базируется на числовых значениях символов. Если строка str2 (по числовому значению) равна строке str1, но короче ее, то считается, что строка str2 меньше строки str1 и наоборот.

lexcmp (str1, str2) char *str1, *str2;

Эта функция Смолл-Си подобна функции strcmp, за тем исключением, что производится лексикографическое сравнение. Чтобы результат имел смысл, строки должны содержать только символы кода ASCII (в диапазоне десятичных значений 0...127). Буквы сравниваются в алфавитном порядке; при этом строчные буквы заменяются соответствующими им прописными. Считается, что буквам предшествуют специальные символы, а им, в свою очередь, предшествуют управляющие символы, за исключением символа DEL, которому приписывается наибольшее значение.

strncmp (str1, str2, n) char *str1, *str2; int n;

Эта функция действует подобно функции strcmp, за исключением того, что сравнивается максимум n символов.

strcpy (dest, sour) char *dest, *sour;

Эта функция копирует строку sour в строку dest и возвращает строку dest. Область, отведенная для строки dest, должна быть достаточно большой, чтобы вместить строку sour.

strncpy (dest, sour, n) char *dest, *sour; int n;

Эта функция действует подобно функции strcpy, за исключением того, что в строку назначения помещается n символов независимо от длины исходной строки. Если исходная строка слишком короткая, то в строке назначения она дополняется нулями, если же слишком длинная, то не уместившиеся символы отбрасываются. После последнего символа в строку назначения записывается нулевой символ.

strlen (str) char *str;

Эта функция возвращает число символов в строке str. Нулевой символ, заканчивающий строку, в это число не включается.

strchr (str, c) char *str, c;

Эта функция возвращает указатель на первый символ с в строке str. Если символ не найден, она возвращает символ NULL. Поиск заканчивается на первом нулевом символе.

strrchr (str, c) char *str, c;

Эта функция действует подобно функции strchr, за исключением того, что осуществляется поиск самого правого символа c.

reverse (str) char *str;

Эта функция меняет на обратный порядок символов в строке str, заканчивающейся нулевым символом.

ФУНКЦИИ КЛАССИФИКАЦИИ СИМВОЛОВ

Следующие функции определяют, относится ли некоторый символ к заданному классу символов. Они возвращают значение *истина* (не нуль) в случае положительного ответа и *ложь* (нуль) в противном случае.

isalnum (c) char c;

Эта функция определяет, является символ с буквой (A-Z или a-z) или цифрой (0...9).

isalpha (c) char c;

Эта функция определяет, является ли символ с буквой (A-Z или a-z).

isascii (c) char c;

Эта функция определяет, является ли символ с символом кода ASCII (десятичные значения 0...127).

iscntr (c) char c;

Эта функция определяет, является ли символ с управляющим символом (значения кода ASCII 0...31 или 127).

isdigit (c) char c;

Эта функция определяет, является ли символ с цифрой (0...9).

isgraph (c) char c;

Эта функция определяет, является ли символ с графическим символом (значения кода ASCII 33...126).

islower (c) char c;

Эта функция определяет, является ли символ с строчной буквой (значения кода ASCII 97...122).

isprint (c) char c;

Эта функция определяет, является ли символ с печатаемым символом (значения кода ASCII 32...126).

ispunct (c) char c;

Эта функция определяет, является ли символ с знаком пунктуации (все коды ASCII за исключением управляющих символов, букв и цифр).

isspace (c) char c;

Эта функция определяет, является ли символ с символом-разделителем (символы кода ASCII SP, HT, VT, CR, LF или FF).

isupper (c) char c;

Эта функция определяет, является ли символ с прописной буквой (значения кода ASCII 65...90).

isxdigit (c) char c;

Эта функция определяет, является ли символ с шестнадцатеричной цифрой (0...9, A-F или a-f).

lexorder (c1, c2) char c1,c2;

Эта функция Смолл-Си возвращает целое число, которое меньше, равно или больше нуля, в зависимости от лексикографического соотношения символов c1 и c2, . Чтобы результат имел смысл, должны передаваться только символы, входящие в набор символов кода ASCII (десятичные значения 0...127). Буквы сравниваются в алфавитном порядке; при этом строчные буквы заменяются соответствующими им прописными. Считается, что буквам предшествуют специальные символы, а им, в свою очередь, предшествуют управляющие символы, за исключением символа DEL, которому присваивается наибольшее значение.

ФУНКЦИИ ПРЕОБРАЗОВАНИЯ СИМВОЛОВ

toascii (c) char c;

Эта функция возвращает эквивалент символа c в коде ASCII. В системах, где используется код ASCII, она возвращает символ c без изменения. Эта функция позволяет применять в программах код ASCII независимо от реализации компилятора Смолл-Си.

tolower (c) char c;

Если символ c - прописная буква, то эта функция возвращает эквивалентную ей строчную; иначе символ c возвращается без изменения.

toupper (c) char c;

Если символ c - строчная буква, то эта функция возвращает эквивалентную ей прописную; иначе символ c возвращается без изменения.

МАТЕМАТИЧЕСКИЕ ФУНКЦИИ

abs (nbr) int nbr;

Эта функция возвращает абсолютное значение числа nbr.

sign (nbr) int nbr;

Эта функция возвращает -1, 0 или 1, в зависимости от того, меньше, равно или больше нуля число nbr.

ФУНКЦИИ УПРАВЛЕНИЯ ПРОГРАММОЙ

calloc (nbr,sz) int nbr, sz;

Эта функция выделяет блок памяти объемом nbr*sz байтов и инициализирует его, заполняя нулями. При успешном выполнении функция возвращает адрес этого блока памяти; в противном случае она возвращает нуль.

malloc (nbr) int nbr;

Эта функция выделяет неинициализированный блок памяти объемом nbr байтов. При успешном выполнении она возвращает адрес этого блока памяти; в противном случае она возвращает нуль.

avail (abort) int abort;

Эта функция возвращает число байтов свободной памяти между программой и стеком. Она проверяет также, не попадает ли стек в выделенную область памяти; если это произошло и параметр abort не равен нулю, то работа программы прекращается и на консоль выдается буква S, указывая тем самым, что произошла ошибка, связанная со стеком. Однако, если параметр abort равен нулю, вызывающей программе возвращается нуль. Функция avail позволяет полностью использовать всю доступную память. Надо, однако, быть внимательным и оставлять достаточный объем памяти для стека.

free (addr) char *addr;

cfree (addr) char *addr;

Эти функции освобождают блок памяти, начиная с адреса addr. При успешном выполнении они возвращают адрес addr; иначе - символ NULL. В большинстве реализаций компилятора

Смолл-Си память выделяется, начиная с конца программы, в виде смежных блоков и вся память освобождается, начиная с адреса addr. Следовательно, освобождать память надо в порядке, обратном тому, в котором она выделялась. Следует избегать освобождения памяти, выделенной до открытия файла, так как можно ожидать, что функция открытия динамически выделяет буфер памяти. Нет нужды говорить о том, что нежелательно освобождать, а потом вновь выделять память для буферов файла. Не следует также рассчитывать на то, что при закрытии файла происходит отказ от буфера, так как в зависимости от значения описателя файлов fd буфер может резервироваться для будущего использования.

getarg (nbr, str, sz, argc, argv)

char *str; int nbr, sz, argc, *argv;

Эта функция Смолл-Си пересылает аргумент командной строки, задаваемый номером nbr, в строку str с максимальной длиной sz и возвращает длину полученного поля. Сразу за аргументом в строку записывается нулевой байт. Аргументы argc и argv должны быть теми же самыми, что и для функции main в начале программы. Для получения имени программы номер nbr должен быть равен нулю, для получения первого аргумента, следующего за именем программы, - единице, и т.д. В реализации компилятора Смолл-Си для ОС CP/M нельзя получить имя программы, поэтому вместо него может быть подставлено некоторое фиксированное значение (например, звездочка). Если нет аргумента, соответствующего номеру nbr, то функция getarg записывает в строку str нулевой байт и возвращает символ EOF.

poll (pause) int pause;

Эта функция Смолл-Си опрашивает ввод с консоли. Если не было ввода, то возвращается нуль. Если был введен какой-либо символ, то дальнейшие действия определяются значением аргумента pause. Если оно равно нулю, то символ немедленно возвращается вызывающей программе. Если аргумент pause - не нуль и введен управляющий символ Ctrl-S, то выполнение программы приостанавливается; при вводе следующего символа вызывающей программе возвращается нуль. Если введен управляющий символ Ctrl-C, то выполнение программы заканчивается. Все остальные символы немедленно возвращаются вызывающей программе.

abort (errcode) int errcode;

exit (errcode) int errcode;

Эти функции закрывают все открытые файлы и возвращают управление операционной системе. Действия, определяемые значением аргумента errcode, зависят от реализации компилятора Смолл-Си. В простейшем случае значение аргумента errcode может быть выведено на консоль. Например, если программа завершается управляющим символом Ctrl-G, то на консоли звучит сигнал.

ГЛАВА 18

ГЕНЕРАЦИЯ КОДА

Цель этой главы - дать более глубокое понимание работы компилятора с помощью сравнения операторов языка Си и генерируемого для них кода на языке ассемблера. Тайны нового языка лучше всего раскрываются в процессе тщательного изучения конечного продукта работы компилятора. В гл.2 представлены основные понятия языка ассемблера, а в гл.3 описана система команд микропроцессора 8080. При изучении приведенных ниже примеров следует пользоваться материалом этих глав.

По умолчанию компилятор вводит исходный текст из стандартного файла ввода и записывает результаты своей работы в стандартный файл вывода. Это существенно облегчает изучение генерируемого компилятором кода. Если его вызвать, не задавая файл ввода и не переадресуя ввод или вывод, то все, что поступает с клавиатуры, становится для компилятора входной информацией, а код, который он генерирует, появляется на экране. Надо просто вводить какие-либо операторы и смотреть, что при этом выдает компилятор.

Прежде всего необходимо определить несколько понятий, лежащих в основе того, чем ЦП является для компилятора. С точки зрения компилятора ЦП включает в себя два 16-разрядных регистра: *первичный* регистр и *вторичный* регистр. Первичный регистр является основным аккумулятором значений выражений и подвыражений. При выполнении бинарной операции (например, сложения) левый операнд помещается во вторичный регистр, а правый операнд загружается в первичный регистр. Затем выполняется операция, и результат поступает в первичный регистр. Когда компилятор работает с микропроцессором 8080, в качестве первичного регистра он использует пару регистров HL, а в качестве вторичного - пару регистров DF. Пара регистров BC используется только для чтения из стека ненужных значений и записи в стек локальных переменных; таким образом, содержимое регистров BC не существенно.

Считается также, что механизм работы ЦП со стеком подобен используемому в микропроцессоре 8080. Следовательно, компилятор использует регистр SP микропроцессора 8080. Предполагается также, что существует библиотека арифметических и логических подпрограмм, выполняющих в процессе работы программы большинство операций по вычислению выражений. Кроме того,

должны быть подпрограммы для выполнения различных вспомогательных операций (например, чтение и запись операндов, анализ условий в операторах switch). Эти подпрограммы приведены в приложении Б.

КОНСТАНТЫ

В листинге 18.1 показаны примеры двух функций, каждая из которых содержит набор константных выражений, составляющих самостоятельные операторы. Заметим, что каждый оператор загружает значение в первичный регистр. Простые числовые и символьные константы становятся непосредственными операндами команды

```
LXI H, ...
```

Отметим также, что константные выражения типа

```
123+321
```

вычисляются компилятором и результат загружается как одна константа.

```
На входе                    На выходе

func1() {
                            FUNC1::
123;
                            LXI H,123
123 + 321;
                            LXI H,444
"abc";
                            LXI H,CC2+0
"def";
                            LXI H,CC2+4
}
                            RET
                            CC2:DB 97,98,99,0,100,101,102,0
func2() {
                            FUNC2::
'a';
                            LXI H,97
'\1\';
                            LXI H,257
"ghi";
                            LXI H,CC3+0
}
                            RET
                            CC3:DB 103,104,105,0
```

Листинг 18.1. Код, генерируемый константными выражениями

В заключение заметим, что символьная строка генерирует адрес символьного массива, заканчивающегося нулевым байтом. Этот массив компилятор помещает в конец функции, содержащей строку. Значение адреса будет вычислено ассемблером как смещение относительно метки, сгенерированной компилятором. То, что эти символьные строки лежат между функциями, никак не влияет на выполнение программы, поскольку все выполняемые операторы находятся внутри функций.

ОПИСАНИЯ ГЛОБАЛЬНЫХ ОБЪЕКТОВ И ССЫЛКИ НА НИХ

Примеры, приведенные в листинге 18.2, иллюстрируют генерацию кода для описаний глобальных объектов. Описания целых переменных приведены слева, а символьных - в соответствующих строках справа; таким образом, их легко сравнивать. Заметим, что команды DW (определить слово) определяют целые, а команды DB (определить байт) определяют символы.

```
На входе   На выходе          На входе        На выходе

int                           char
  gi,                           gc,
        GI::                            GC::
        DW 0                            DB 0

  gi2 = 123,                    gc2 = 'a',
        G12::                           GC2::
        DW 123                          DB 97

gia[10] = {1,2, 3},           gca[10] = "abc",
        GIA::                           GCA::
        DW 1,2,3                        DB 97,98,99,0
        DW 0,0,0,0,0,0,0                DB 0,0,0,0,0,0

*gip;                         *gcp;
        GIP::                           GCP::
        DW 0                            DW 0
```

Листинг 18.2. Код, генерируемый глобальными объектами

Эти примеры хорошо иллюстрируют результат инициализации данных. Заметим, что, когда число начальных данных меньше числа элементов, первые элементы инициализируются, а остальные обнуляются. (*Примечание.* Имена объектов в этих примерах выбирались таким образом, чтобы они указывали на их вид. Например, gi - *глобальное целое*, gca - *глобальный символьный массив*, gip - *глобальный указатель на целое*. В следующих примерах используются те же самые соглашения; при этом l означает *локальный*, a - *аргумент* (*параметр*), e - *внешний*.)

Примеры, приведенные в листинге 18.3, иллюстрируют генерацию кода при ссылках на глобальные объекты. Примеры для целых и символов приведены в левом и правом столбцах. Подпрограммы CCSXT, CCGINT и CCGCHAR находятся в библиотеке арифметических и логических подпрограмм (см. приложение Б). Чтобы получить полное представление о том, как реализуются ссылки, следует ознакомиться с этими подпрограммами. Приложение Б будет полезно и для понимания остальных примеров.

```
На входе    На выходе          На входе    На выходе
gi;                            gc;
            LHLD GI                        LDA GC
                                           CALL CCSXT
&gi;                           &gc;
            LXI H,GI                       LXI H,GC
gia;                           gca;
            LXI H,GIA                      LXI,H,GCA
gia[0];                        gca[0];
            LXI H,GIA                      LXI H,GCA
            CALL CCGINT                    CALL CCGCHAR
*gia;                          *gca;
            LXI H,GIA                      LXI H,GCA
            CALL CCGINT                    CALL CCGCHAR
gia[5];                        gca[5];
            LXI H,GIA                      LXI H,GCA
            LXI D,10                       LXI D,5
            DAD D                          DAD D
            CALL CCGINT                    CALL CCGCHAR
*(gia + 5);                    *(gca + 5);
            LXI H,GIA                      LXI H,GCA
            LXI D,10                       LXI D,5
            DAD D                          DAD D
            CALL CCGINT                    CALL CCGCHAR
gip;                           gcp;
            LHLD GIP                       LHLD GCP
*gip;                          *gcp;
            LHLD GIP                       LHLD GCP
            CALL CCGINT                    CALL CCGCHAR
gip[5];                        gcp[5];
            LHLD GIP                       LHLD GCP
            LXI D,10                       LXI D,5
            DAD D                          DAD D
            CALL CCGINT                    CALL CCGCHAR
*(gip + 5);                    *(gcp + 5);
            LXI H,GIP                      LXI GCP
            LXI D,10                       LXI D,5
            DAD D                          DAD D
            CALL CCGINT                    CALL CCGCAR
```

Листинг **18.3**. Код, генерируемый глобальными ссылками

Заметим, что ссылки на указатели массивов без индексов относятся к адресам, а не к объектам, находящиеся по этим адресам. Можно увидеть также результат генерации кода для

операций получения адреса и обращения по адресу. В примерах с использованием индексов показано, что для целых смещение удваивается, чтобы учесть, что целые занимают по 2 байта. То же самое справедливо и при сложении каких-либо значений с элементами целых массивов или указателями. Заметим также, что как массивы, так и указатели могут иметь индексы. Различие состоит в том, что адреса массивов получают с помощью команды

```
LXI H, ...
```

а указатели обрабатываются в качестве операндов с помощью команды

```
LHLD H, ...
```

ОПИСАНИЯ ВНЕШНИХ ОБЪЕКТОВ И ССЫЛКИ НА НИХ

Примеры в листинге 18.4 иллюстрируют генерацию кода в случае, когда объекты объявляются внешними. Заметим, что для этих объектов нет определений, они просто объявляются внешними для ассемблера.

```
На входе    На выходе          На входе    На выходе
extern int                     extern int
  ei1                            ec1
            EXT EI1                        EXT EC1
ei2[10];                         ec2[10];
            EXT EI2                        EXT EC2
```

Листинг **18.4**. Код, генерируемый внешними объявлениями

Примеры в листинге 18.5 показывают, что код, генерируемый при ссылках на внешние объекты, не отличается от кода обычных глобальных объектов.

```
На входе    На выходе          На входе    На выходе
ei1;                           ec1;
            LHLD EI1                       LDA EC1
                                           CALL CCSXT
ei2;                           ec2;
            LXI H,EI2                      LXI H,EC2
ei2[5];                        ec2[5];
            LXI H,EI2                      LXI H,EC2
            LXI D,10                       LXI D,5
            DAD D                          DAD D
            CALL CCGINT                    CALL CCGCHAR
```

Листинг **18.5**. Код, генерируемый внешними ссылками

ОПИСАНИЯ ЛОКАЛЬНЫХ ОБЪЕКТОВ И ССЫЛКИ НА НИХ

Функции в листинге 18.6 иллюстрируют, как описываются локальные объекты и осуществляются ссылки на них. В листинге такие функции начинаются в левом столбце и продолжаются до конца правого столбца. Кроме того, слева даны ссылки на целые переменные, а справа - на символьные.

```
На входе    На выходе
func () {
            FUNC::
char lc,    lca[10], *lcp;
int li,     lia[10], *lip;
li;
            LXI H,-37
            DADSP
            SPHL

            LXI H,22
            DAD SP
            CALL CCGINT
lia;
            LXI H,2
            DAD SP
lia[0];
            POP B
            POP H
            PUSH H
            PUSH B
*lia;
            POP B
            POP H
            PUSH H
            PUSH B
lia[5];
            LXI H,2
            DAD SP
            LXI D,10
            DAD D
            CALL CCGINT

*(lia + 5);
            LXI H,2
            DAD SP
            LXI D,10
            DAD D
            CALL CCGINT
lip;
            POP H
            PUSH H
lip[0];
            POP H
            PUSH H
            CALL CCGINT
*lip;
            POP H
            PUSH H
            CALL CCGINT
lip[5];
            POP H
            PUSH H
            LXI D,10
            DAD D
            CALL CCGINT
*(lip + 5);
            POP H
            PUSH H
            LXI D,10
            DAD D
            CALL CCGINT
```

```
На входе    На выходе
lc;
            LXI H,36
            DAD SP
            CALL CCGCHAR
lca;
            LXI H,26
            DAD SP
lca[0];
            LXI H,26
            DAD SP
            CALL CCGCHAR
*lca;
            LXI H,26
            DAD SP
            CALL CCGCHAR
lca[5];
            LXI H,26
            DAD SP
            LXI D,5
            DAD D
            CALL CCGCHAR

*(lca + 5);
            LXI H,26
            DAD SP
            LXI D,5
            DAD D
            CALL CCGCHAR
lcp;
            LXI H,24
            DAD SP
            CALL CCGINT
lcp[0];
            LXI H,24
            DAD SP
            CALL CCGINT
            CALL .CCGCHAR
*lcp;
            LXI H,24
            DAD SP
            CALL CCGINT
            CALL CCGCHAR
lcp[5];
            LXI H,24
            DAD SP
            CALL CCGINT
            LXI D,5
            DAD D
            CALL CCGCHAR
*(lcp + 5);
            LXI H,24
            DAD SP
            CALL CCGINT
            LXI D,5
            DAD D
            CALL CCGCHAR
}
            LXI H,37
            DAD SP
            SPHL
            RET
```

Листинг 18.6. Код, генерируемый локальными объектами/ссылками

Прежде всего отметим, что память для всех локальных переменных выделяется одновременно в тот момент, когда появляется первый выполняемый оператор данного блока. Это делается с помощью добавления к указателю стека отрицательного числа. Кроме того, заметим, что ссылки на локальные объекты производятся относительно вершины стека, для чего необходимо загрузить константу и затем прибавить ее к значению указателя стека. И в заключение для получения желаемого объекта, адрес которого содержится в регистрах HL, вызывается библиотечная подпрограмма.

Исключением из этого правила является случай, когда целое или указатель находится в вершине стека или в соседнем слове. Тогда с помощью простой последовательности команд чтение/запись получают 16-разрядный объект, оставляя без изменения начальное значение указателя стека. Следующий объект получают с помощью последовательности чтение/чтение/запись/запись. Заметим, что в этом случае используется пара регистров BC, так как в них не содержится ничего важного.

Примеры в листинге 18.7 показывают генерацию кода при описании функции с аргументами, передаче аргументов функции и ссылках на аргументы. Ссылки на целые находятся слева, а на символы - справа.

```
На входе    На выходе                  На входе    На выходе

func1(ai, aia, aip, ac, aca, acp)
        FUNC1::
    int  ai, aia[], *aip;
    char ac, aca[], *acp;
    {
ai;                                    ac;
        LXI H,12                               LXI H,6
        DAD SP                                 DAD SP
        CALL CCGINT                            CALL CCGCHAR
aia;                                   aca;
        LXI H,10                               LXI H,4
        DAD SP                                 DAD SP
        CALL CCGINT                            CALL CCGINT
aia[5];                                aca[5];
        LXI H,10                               LXI H,4
        DAD SP                                 DAD SP
        CALL CCGINT                            CALL CCGINT
        LXI D,10                               LXI D,5
        DAD D                                  DAD D
        CALL CCGINT                            CALL CCGCHAR
aip;                                   acp;
        LXI H,8                                POP B
        DAD SP                                 POP H
        CALL CCGINT                            PUSH H
                                               PUSH B
*aip;                                  *acp;
        LXI H,8                                POP B
        DAD SP                                 POP H
        CALL CCGINT                            PUSH H
        CALL CCGINT                            PUSH B
                                               CALL CCGCHAR
                                   return (gi1);
                                               LHLD GI1
                                               RET
                                       }
```

Листинг 18.7. Код, генерируемый аргументами/ссылками функций

Из этих примеров видно, что, хотя ссылки на аргументы, как и на локальные переменные, организуются через стек, между этими двумя случаями существуют некоторые отличия.

Первое и самое главное отличие состоит в том, что функция не выделяет память под аргументы. Память под них отводится при вызове функции. Считается, что, когда функция получает управление, аргументы уже находятся в стеке. Если при вызове была допущена ошибка в числе или типе аргументов, то все равно считается, что аргументы были переданы так, как это должно быть. Неправильная передача аргументов является распространенной ошибкой, приводящей к непредсказуемому и, весьма возможно, неуправляемому поведению программы. Это первое, на что должно падать подозрение, когда программа "выходит из повиновения".

Другое отличие состоит в том, что при входе в функцию аргументы не расположены в вершине стека. Это место занято адресом возврата, используемым для передачи управления в точку, следующую за вызовом функции. Заметим, что ссылка на аргумент acp, последний аргумент, записанный в стек, генерирует последовательность команд обращения к стеку чтение/чтение/запись/запись.

Последним оператором функции является оператор return. Если бы он отсутствовал, то все равно была бы сгенерирована команда RET. Кроме того, если были описаны локальные переменные, то до выполнения команды RET должно быть освобождено место в стеке, занятое под эти переменные.

Пример в листинге 18.8 показывает генерацию кода при вызове функции. Последовательность команд здесь очень простая. Во-первых, каждый аргумент вычисляется и записывается в стек. Затем в регистр A загружается число аргументов. После этого выполняется вызов функции. И в конце, после возврата из функции, освобождается память, занятая в стеке аргументами.

```
На входе                    На выходе
func(gi, gia, gip);
                            LHLD GI
                            PUSH H
                            LXI H,GIA
                            PUSH H
                            LHLD GIP
                            PUSH H
                            MVI A,3

                            CALL FUNC

                            POP B
                            POP B
                            POP B
```

Листинг 18.8. Код, генерируемый прямыми вызовами функций

Счетчик аргументов дает возможность определить в функции число переданных аргументов. Так как регистр A постоянно используется и легко может измениться, счетчик аргументов следует получать и запоминать сразу после входа в функцию. Это можно прекрасно сделать с помощью оператора

```
count=CCARGC();
```

где идентификатор CCARGC обозначает точку входа в динамической рабочей библиотеке; эта функция возвращает в первичном регистре содержимое регистра A в виде 16-разрядного целого числа.

Функции, использующие счетчик аргументов, без изменений в другие компиляторы перенести нельзя, так как в них предполагается знание того, как передаются аргументы в Смолл-Си.

Во многих программах нет нужды в передаче счетчика аргументов, тем более что для этого при вызове функции требуются дополнительные команды. Поэтому, поместив в начало программы команду

```
#define NOCCARGC
```

можно отказаться от передачи значения счетчика аргументов. Тогда программа после компиляции не будет содержать команд, передающих значение счетчика. Этот счетчик используется только в четырех стандартных функциях: printf, fprintf, scanf и fscanf, каждой из которых передается управляющая строка, за которой следует не определенное заранее число аргументов. Эти функции определяют число обрабатываемых аргументов, просматривая управляющую строку. Однако для того, чтобы найти управляющую строку (первый аргумент), нужно знать, сколько аргументов было передано. Если в программе, использующей хотя бы одну из перечисленных выше функций, определить имя NOCCARGC, то это будет ошибкой.

Если адрес функции вычисляется, а не получается прямо по метке функции, то требуется другой вид вызова функции. Эта ситуация иллюстрируется примером в листинге 18.9.

Здесь заметим, что адрес функции (десятичное число 256) записывается в стек. Затем, после вычисления каждого аргумента, используется команда XTHL, обменивающая полученное значение на значение, находящееся в вершине стека. Если есть еще аргумент, то адрес функции снова записывается в стек. Таким образом, во время обработки аргументов соответствующий адрес все время находится в вершине стека. Далее, как обычно, в регистр A загружается счетчик аргументов. И наконец, вместо вызова функции происходит вызов функции CCDCAL, которая находится в библиотеке арифметических и логических подпрограмм, содержащей лишь одну команду PCHL (вспомним по гл.3, что это переход *по адресу, содержащемуся в регистрах HL*).

На входе	На выходе
256 (gi, gc);	
	LXI H,256
	PUSH H
	LHLD gi
	XTHL
	PUSH H
	LDA gc
	CALL CCSXT##
	XTHL
	MVI A,2
	CALL CCDCAL##
	POP B
	POP B

Листинг 18.9. Код, генерируемый косвенными вызовами функций

Поскольку для выполнения команды PCHL была использована команда CALL, то в стек был записан адрес возврата, так что при возврате из подпрограммы, находящейся по адресу 256, управление передается первой из двух команд POP B.

ВЫРАЖЕНИЯ

Можно продолжить демонстрацию примеров интересных выражений. Однако для краткости будут проиллюстрированы только типичные образцы выражений для различных операций и один пример с достаточно сложным выражением. Кроме того, для упрощения будут использоваться только глобальные объекты.

В примерах листингов 18.10 и 18.11 показаны результаты выполнения унарных операторов. Сначала рассмотрим логическую операцию НЕ. Как вы можете видеть, операция НЕ реализуется как вызов подпрограммы CCLNEG, находящейся в библиотеке арифметических и логических подпрограмм. Эта подпрограмма выполняет операцию логического отрицания над значением, содержащимся в регистрах HL, и возвращает результат в тех же регистрах.

На входе	На выходе
!gi;	
	LHLD GI
	CALL CCLNEG

Листинг 18.10. Код, генерируемый логическим оператором НЕ

Операция увеличения (листинг 18.11) добавляет единицу к глобальной целой переменной gi и затем записывает новое значение по адресу хранения переменной gi.

```
На входе                На выходе
++gi;
                        LHLD GI
                        INX H
                        SHLD GI
```

Листинг 18.11. Код, генерируемый оператором увеличения, стоящим перед операндом

Теперь по листингу 18.12 проследим, что произойдет, если в предыдущий листинг внести два изменения. При задании в выражении указателя на целое, а не самого целого значение указателя увеличится на 2, так что он будет переустановлен на следующее целое. Отметим также, что так как знак операции увеличения стоит справа от операнда, после обновления памяти исходное значение операнда в регистрах HL восстанавливается.

```
На входе                На выходе
gip++;
                        LHLD GIP
                        INX H
                        INX H
                        SHLD GIP
                        DCX H
                        DCX H
```

Листинг 18.12. Код, генерируемый оператором увеличения, стоящим после операнда

В листинге 18.13 операция обращения по адресу применяется к целому указателю gip один, два и три раза. В первом случае

```
На входе                На выходе
*gip;
                        LHLD GIP
                        CALL CCGINT
**gip;
                        LHLD GIP
                        CALL CCGINT
                        CALL CCGINT
***gip;
                        LHLD GIP
                        CALL CCGINT
                        CALL CCGINT
                        CALL CCGINT
```

Листинг 18.13. Код, генерируемый оператором косвенности

16-разрядное значение, на которое указывает gip, выбирается с помощью подпрограммы CCGINT, загружающей в регистры HL целое, заданное содержимым регистров HL. Во втором случае получается 16-разрядное значение, на которое указывает полученное ранее значение. В третьем случае применяется еще один уровень косвенности.

Операция получения адреса (листинг 18.14) вызывает загрузку в регистры HL адреса gip. Заметим, что при этом загружается адрес gip, а не адрес, находящийся в ячейке gip. Операция получения адреса элемента массива gia[5] генерирует тот же самый код, что и для ссылки на этот элемент, но без вызова подпрограммы CCGINT.

```
На входе                На выходе
&gip;
                        LXI H,GIP
&gia[5];
                        LXI H,GIA
                        LXI D,10
                        DAD D
```

Листинг 18.14. Код, генерируемый оператором адреса

Операции деления и деления по модулю (листинг 18.15) генерируют идентичные коды, за тем исключением, что для операции деления по модулю в конце добавляется команда XCHG. Так как подпрограмма CCDIV возвращает частное в регистрах HL, а остаток в регистрах DE, команда XCHG подставляет остаток на место частного. (*Примечание.* Необычные двойные точка с запятой после команды XCHG вызваны тем, что компилятор убрал команду PUSH H, когда стало очевидно, что нет необходимости в последовательности команд обращения к стеку *запись/чтение.* Эти точки с запятой воспринимаются ассемблером как комментарии.)

```
На входе                На выходе
gi / 5;
                        LHLD GI
                        XCHG;;
                        LXI H,5
                        CALL CCDIV
gi % 5;
                        LHLD GI
                        XCHG;;
                        LXI H,5
                        CALL CCDIV
                        XCHG;;
```

Листинг 18.15. Код, генерируемый операторами деления и получения остатка

Операция сложения (листинг 18.16) интересна тем, что для ее реализации достаточно одной команды DAD D. Никакие подпрограммы при этом не вызываются.

```
На входе            На выходе
gi +5;
                    LHLD GI
                    LXI D,5
                    DAD D
```

Листинг 18.16. Код, генерируемый оператором сложения

Операции отношения, проиллюстрированные в листинге 18.17 операцией равенства, не сильно отличаются от других бинарных операций. Для выполнения сравнения вызывается библиотечная подпрограмма, в данном случае подпрограмма CCEQ. Если она устанавливает значение *истина*, то в регистрах HL возвращается единица; иначе возвращается нуль.

```
На входе            На выходе
gi == 5;
                    LHLD GI
                    XCHG;;
                    LXI H,5
                    CALL CCEQ
```

Листинг 18.17. Код, генерируемый оператором равенства

Логическая операция И (листинг 18.18) интересна тем, что при выполнении нескольких таких операций первое же значение *ложь* (нуль) заканчивает процесс проверки (для логической операции ИЛИ все происходит наоборот: проверка заканчивается при первом же значении *истина*). Заметим, что эти последовательности не обязательно должны содержать операции отношения. Так, последний терм gc дает свое собственное значение, которое считается значением *истина* при неравенстве его нулю и значением *ложь* в противоположном случае.

Операции присваивания, приведенные в листинге 18.19, являются типичными для всех операций присваивания. Сначала осуществляется прямое присваивание, а затем - присваивание +=. Заметим, что присваивание += оказывает тот же самый эффект, что и выражение

```
gi = gi + 5
```

Заметим также, что константа 5 в этом случае прямо загружается в регистры DE (вместо загрузки в регистры HL и обмена с регистрами DE). Здесь компилятор "видит", что регистры DE не потребуются для сложных вычислений справа от знака операции.


```
На входе                    На выходе
gi > 1 && gi < 5 && gc;
                            LHLD GI
                            XCHG;;
                            CALL CCGT
                            MOV A,H
                            ORA L
                            JZ CC3

                            LHLD GI
                            XCHG;;
                            LXI H,5
                            CALL CCLT
                            MOV A,H
                            ORA L
                            JZ CC3

                            LDA GC
                            CALL CCSXT
                            MOV A,H
                            ORA L
                            JZ CC3

                            LXI H,1
                            JMP CC4
                        CC3:
                            LXI H,0
                        CC4:
```

Листинг 18.18. Код, генерируемый логическим оператором И

```
На входе            На выходе
gi = 5
                    LXI H,5
                    SHLD GI
gi += 5;
                    LHLD GI
                    LXI D,5
                    DAD D
                    SHLD GI
```

Листинг 18.19. Код, генерируемый операторами присваивания

Последний пример (листинг 18.20) объединяет рассмотренные выше концепции и, кроме того, иллюстрирует возможности комбинирования операций всех типов. В приведенном в листинге выражении сначала вызывается функция func и возвращаемое ею значение присваивается переменной gc. Затем это значение срав-

нивается со значением 5, в результате чего получается логическое значение *истина* (единица) или *ложь* (нуль), которое, в свою очередь, умножается на 'y', давая значение, присваиваемое переменной gi. Таким образом, если функция возвращает значение 5, то переменная gi устанавливается равной значению символа y в коде ASCII; иначе переменной gi присваивается значение нуль. В любом случае значение переменной gc станет равным значению, возвращаемому функцией func.

На входе	На выходе
gi = 'y' * ((gc = func()) == 5);	XRA A CALL FUNC MOV A,L STA GC XCHG;; LXI H,5 CALL CCEQ LXI D,121 CALL CCMULT SHLD GI

Листинг 18.20. Код, генерируемый сложным выражением

ОПЕРАТОРЫ

Ниже приведены примеры генерации кода для всех операторов, известных компилятору. Сначала рассмотрим простой оператор if (листинг 18.21).

На входе	На выходе
if (gi) gi = 5;	LHLD GI MOV A,H ORA L JZ CC3 LXI H,5 SHLD GI CC3:

Листинг 18.21. Код, генерируемый оператором IF

Проверяемое выражение состоит только из одной переменной. Эта переменная загружается в регистры HL, а затем проверяется, для чего старший байт пересылается в регистр A, и выполняется логическая операция ИЛИ для этого байта и младшего байта. Если результат - *ложь* (нуль), то оператор, управляемый оператором if, не выполняется. В противном случае результат имеет значение *истина*, и выполняется оператор


```
gi = 5;
```

В оператор if в листинге 18.22 входит условие else. Если переменной gi соответствует значение *истина*, то выполняется первый управляемый оператор

```
gi = 5;
```

если же получено значение *ложь*, то выполняется второй оператор

```
gi = 10;
```

Таким образом, выполняется один, и только один, из этих операторов. Заметим, что в конце первого оператора выполняется обход второго оператора.

На входе	На выходе
if (gi) gi = 5; else gi = 10;	LHLD GI MOV A,H ORA L JZ CC4 LXI H,5 SHLD GI JMP CC5 CC4: LXI H,10 SHLD GI CC5

Листинг 18.22. Код, генерируемый оператором IF/ELSE

Два оператора if в листинге 18.23 иллюстрируют еще большую эффективность генерации кода при проверке на равенство или неравенство нулю. Обратим внимание на константы, помеченные звездочками. В первом случае константа загружается в регистры HL, а затем вызывается библиотечная подпрограмма. Эти команды занимают больше места и выполняются дольше (что связано в основном с тем, что должна выполняться подпрограмма CCNE), чем в основном примере.

Оператор switch является самым сложным оператором языка Смолл-Си, но разобраться в нем нетрудно. Типичный оператор switch показан в листинге 18.24. Заметим, что непосредственным эффектом действия префиксов case и default является то, что они сразу генерируют метку на языке ассемблера. Позже, в конце оператора, для каждого префикса case генерируется пара слов. Эти слова содержат адрес (метку) для данного префикса и значение выражения в операторе switch, при котором управление передается по этому адресу. Сначала вычисляется выражение gc, и его значение помещается в регистры HL. Затем выполняется об-

```
На входе                    На выходе
if (gi != 1) gi = 5;
                                LHLD GI
                                XCHG;;
                            *   LXI H,1
                            *   CALL CCNE
                            *   MOV A,H
                            *   ORA L
                                JZ CC9
                                LXI H,5
                                SHLD GI
                            CC9:

if (gi != 0) gi = 5;
                                LHLD GI
                            *   MOV A,H
                            *   ORA L
                                JZ CC10
                                LXI H,5
                                SHLD GI
                            CC10:
```

Листинг **18.23**. Код, генерируемый проверками на неравенство и равенство нулю

ход операторов, управляемых оператором switch, и вызывается подпрограмма CCSWITCH, которая просматривает следующий за ней список, состоящий из пар слов, содержащих адреса и значения, и сравнивает эти значения с содержимым регистров HL. При первом же совпадении подпрограмма CCSWITCH передает управление на соответствующий адрес. Если ни одно из значений не совпало (на это указывает слово в конце списка, содержащее нуль), то управление передается в точку, следующую сразу за списком. Здесь находится команда перехода, создаваемая префиксом default. Если нет префикса default, то нет и перехода, и управление просто передается в точку, следующую за оператором switch. Заметим, что последовательность префиксов case прекрасно подходит для того, чтобы при разных значениях выражения в операторе switch передавать управление в одну и ту же точку. Из данного примера хорошо видно, что управляемые оператором switch операторы могут выполняться один за другим независимо от последующих префиксов case/default. Чтобы прервать этот процесс, нужен оператор break, continue или goto. Оператор break генерирует команду передачи управления на метку, ограничивающую оператор switch. Заметим также, что подобная команда перехода следует за последним управляемым оператором.

Как показано в листинге 18.25, оператор while в языке ассемблера имеет чрезвычайно простую структуру. При каждой итерации значение управляющей переменной gi уменьшается и сравнивается со значением *истина/ложь*. Если ее значение *истина* (не нуль), то управление передается на метку, ограничивающую весь оператор; в противном случае вызывается функция и выполняется переход в начало оператора. Этот цикл продолжается до тех пор, пока значение проверяемого выражения не станет равным значению *ложь*.


```
На входе                    На выходе
switch (gc) {
                            CC18:
                                LDA GC
                                CALL CCSXT
                                JMP CC21
 case 'Y':
                            CC22:
 case 'Y': gcp = "yes";
                            CC23:
                                LXI H,CC2+0
                                SHLD GCP
         break;
                                JMP CC20
 case 'N':
                            CC24:
 case 'n': gcp = "no";
                            CC25:
                                LXI H,CC2+4
                                SHLD GCP
         break;
                                JMP CC20
 default: GCP = "error";
                            CC26:
                                LXI H,CC2+7
                                SHLD GCP
}
                                JMP CC20
                            CC21:
                                CALL CCSWITCH
                                DW CC22,89
                                DW CC23,121
                                DW CC24,78
                                DW CC25,110
                                DW 0
                                JMP CC26
                            CC20:
```

Листинг **18.24**. Код, генерируемый оператором SWITCH

На входе

```
while (--gi) func(gi);
```

На выходе

```
CC33:
    LHLD GI
    DCX H
    SHLD GI
    MOV A,H
    ORA L
    JZ CC34
    LHLD GI
    PUSH H
    MVI A,1
    CALL FUNC
    POP B
    JMP CC33
CC34:
```

Листинг 18.25. Код, генерируемый оператором WHILE

На входе

```
for (gi = 4; gi >= 0; --gi) gia[gi] = 0;
```

На выходе

```
    LXI H,4
    SHLD GI
CC39:
    LHLD GI
    XRA A
    ORA H
    JM CC38
    JMP CC40
CC37:
    LHLD GI
    DCX H
    SHLD GI
    JMP CC39
CC40:
    LXI H,GIA
    XCHG;;
    LHLD GI
    DAD H
    DAD D
    XCHG;;
    LXI H,0
    CALL CCPINT
    JMP CC37
CC38:
```

Листинг 18.26. Код, генерируемый оператором FOR

Оператор for (листинг 18.26) начинается с вычисления первого из трех выражений (или списков выражений), стоящих в скобках. Это инициализирующее выражение. Затем вычисляется и проверяется второе, управляющее выражение. Если оно дает значение *ложь*, то выполняется переход на заключительную метку; иначе происходит переход в тело оператора for. Сначала выполняются операторы, управляемые оператором for, после чего следует переход назад на команды, вычисляющие третье выражение. В этом выражении обычно увеличивается или уменьшается управляющая переменная. После этого происходит переход назад для следующего вычисления управляющего выражения. Этот процесс повторяется до тех пор, пока значение управляющего выражения не примет значение *ложь*.

Пример оператора for в листинге 18.27 интересен как иллюстрация того, что происходит, когда опущены все три выражения

На входе

```
for (;;) {gia[gi] = 0; if (++gi > 5) break;}
```

На выходе

```
CC47:
    JMP CC48
CC45:
    JMP CC47
CC48:
    LXI H,GIA
    XCHG;;
    LHLD GI
    DAD H
    DAD D
    XCHG;;
    LXI H,0
    CALL CCPINT
    LHLD GI
    INX H
    SHLD GI
    XCHG;;
    LXI H,5
    CALL CCGT
    MOV A,H
    ORA L
    JZ CC49
    JMP CC46
CC49:
    JMP CC45
CC46:
```

Листинг 18.27. Код, генерируемый оператором FOR без управляющих выражений

в скобках. Если одно из выражений отсутствует, то на этом месте не генерируется никакой код. Если все три выражения отсутствуют, то действие оператора for будет эквивалентно действию оператора

```
while(1)
```

однако результирующий код будет менее эффективным из-за дополнительных команд обхода. Как показано ниже, в подобных случаях для выхода из цикла используется оператор break.

На входе

```
do gia[gi] = 0; while (--gi);
```

На выходе

```
CC52:
    LXI H,GIA
    XCHG;;
    LHLD GI
    DAD H
    DAD D
    XCHG;;
    LXI H,0
    CALL CCPINT
CC50:
    LHLD GI
    DCX H
    SHLD GI
    MOV A,H
    ORA L
    JZ CC51
    JMP CC52
CC51:
```

Листинг 18.28. Код, генерируемый оператором DO/WHILE

Оператор do/while является по сути дела инвертированным оператором while, в котором управляющее выражение вычисляется последним. Управляющий оператор всегда выполняется хотя бы один раз. Пример генерации кода показан в листинге 18.28.

В качестве примера оператора goto рассмотрим листинг 18.29.

На входе

```
abc:

gi = 5;


goto abc;
```

На выходе

```
CC54:

    LXI H,5
    SHLD GI

    JMP CC54
```

Листинг 18.29. Код, генерируемый оператором GOTO

Здесь записана метка, за которой следуют оператор присваивания и оператор goto, ссылающийся на эту метку. Для указанной метки компилятор генерирует некоторое имя, так как одна и та же метка может быть использована в различных функциях. Если бы это имя уже было использовано, то ассемблер отметил бы ошибку, вызванную повторением метки. Для оператора goto генерируется команда безусловного перехода на соответствующую метку.

ЗАКЛЮЧЕНИЕ

На этом заканчивается рассмотрение вопросов, связанных с генерацией кода компилятором Смолл-Си. Можно было бы рассмотреть и другие ситуации - число возможных вариантов бесконечно. Запуская компилятор без задания имен файлов, легко проверить и другие случаи генерации кода. Весьма полезно изучить, как компилятор работает с указателем стека для вложенных составных операторов с локальными переменными. Можно исследовать также, что происходит, когда выполняется выход с помощью операторов break или continue из вложенных в два или более уровней блоков, содержащих локальные описания. Каждый раз, когда неясно, что компилятор будет делать с конкретным оператором, проще всего "спросить" его самого, т.е. просто запустить компилятор без аргументов в командной строке и затем ввести с клавиатуры оператор, вызывающий сомнение. На экране появится результирующий код ассемблера.

Г Л А В А 19

ЭФФЕКТИВНОСТЬ ПРОГРАММ

В идеале мы хотим, чтобы наши программы были эффективны и хорошо спроектированы. При тщательной разработке до начала кодирования обе эти цели в какой-то степени могут быть достигнуты. Затратив однажды на программу значительные усилия, мы с большой неохотой отказываемся от своей работы и начинаем все сначала. Проявляется наш прагматизм, и мы просто латаем то, что уже имеем. Таким образом, первое правило написания хороших и эффективных программ состоит в том, что сначала надо подумать.

В различные моменты проектирования вы будете вынуждены выбирать между хорошим стилем программирования и эффективностью. Говоря о *хорошем стиле программирования*, я имею в виду основные представления об организации программ в виде самостоятельных подпрограмм, которые легко понимать, поскольку их связь с другими подпрограммами обусловлена лишь побочными эффектами. Подпрограммы должны быть достаточно малы, чтобы

их логика была очевидной. Такой подход делает структуру программ более наглядной и экономит драгоценное время при работе с ними. Под эффективностью я подразумеваю малый размер программ и большую скорость их выполнения - часто две взаимоисключающие цели.

Как правило, хороший стиль программирования более важен, чем эффективность, но бывает и так, что более важно, чтобы программа могла выполняться в ограниченной области памяти и/или выполнялась достаточно быстро, чтобы не отставать от реальных событий. Так что искать компромисс между этими двумя целями предстоит вам, программисту. Все советы, данные в этой главе, направлены только на повышение эффективности. Они не обязательно подходят при всех обстоятельствах, и некоторые из них бросают вызов общепринятым приемам структурного программирования. Однако они могут оказаться небесполезными, когда вы будете решать, как вам лучше написать программу.

ЦЕЛЫЕ И ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ОБХОДЯТСЯ ДЕШЕВЛЕ

Из примеров по генерации кода при ссылках на глобальные и локальные переменные (см. гл.18) должно быть ясно, что код для целых переменных эффективнее как по объему, так и по скорости. Например, при выборке глобальной целой переменной gi генерируется команда

```
LHLD GI
```

которая занимает ровно 3 байта и делает 5 обращений к памяти (3 для команды и 2 для операнда). С локальной целой переменной дело обстоит несколько сложнее. Для выборки локальной целой переменной обычно генерируется последовательность

```
LXI  H,nn
DAD  SP
CALL CCGINT
```

где nn - смещение для переменной в стеке. Эта последовательность занимает 7 байтов и требует 7 обращений к памяти. Но это еще не все. Так как последняя команда вызывает библиотечную подпрограмму CCGINT (см. приложение Б), то требуется еще 9 обращений к памяти. Если целая переменная является последней описанной переменой и, следовательно, лежит в вершине стека, то она выбирается с помощью последовательности

```
POP  H
PUSH H
```

которая занимает только 2 байта и требует 6 обращений к памяти. Если она является второй от конца, то выбирается с помощью последовательности

```
POP  B
POP  H
PUSH H
PUSH B
```

Это дает 4 байта и 12 обращений к памяти.

Ситуация с обработкой символьных переменных ничуть не сложнее. Выборка глобального символа gc, например, генерирует последовательность

```
LDA  GC
CALL CCSXT
```

которая имеет длину 6 байтов и требует 14 обращений к памяти (без учета CCSXT). В то же время выборка локального символа генерирует последовательность

```
LXI  H,nn
DAD  SP
CALL CCGCHAR
```

которая требует 7 байтов и 16 обращений к памяти. В табл.19.1 все эти данные сведены вместе.

Таблица 19.1 Эффективность выборки переменных

Длина последовательности, байт	Число обращений к памяти	Тип описания
2	6	Целая, локальная (последняя)
3	5	Целая, глобальная
4	12	Целая, локальная (вторая от конца)
6	14	Символьная, глобальная
7	16	Символьная, локальная
7	16	Целая, локальная (третья и более от конца)

С записью переменных в память дело обстоит так же. Для записи глобальной переменной генерируется команда

```
SHLD GI
```

что требует 3 байта и 5 обращений к памяти. Для записи локальной переменной генерируется последовательность

```
LXI  H,nn
DAD  SP
XCHG
...
CALL CCPINT
```

где многоточие стоит на месте команд, вычисляющих записываемое в память значение. Эта последовательность требует 8 байтов и 16 обращений к памяти, причем в благоприятном случае, когда

адрес назначения пересылается в регистры DE для библиотечной подпрограммы CCPINT с помощью команды XCHG. Если же регистры DE использовались для получения записываемого в память значения, то вместо команды XCHG была бы сгенерирована последовательность

```
PUSH H
...
POP D
```

и потребовалось бы уже 9 байтов и 20 обращений к памяти.

Запись в память глобального символа генерирует последовательность

```
MOV A,L
STA GC
```

что требует 4 байта и 5 обращений к памяти. Запись локального символа генерирует последовательность

```
LXI  H,nn
DAD  SP
XCHG
...
MOV  A,L
STAX D
```

что дает 7 байтов и 8 обращений к памяти. Если вместо команды XCHG используется последовательность

```
PUSH H
...
POP  D
```

то потребность в памяти увеличивается до 8 байтов, а число обращений к памяти становится равным 13. Эти данные сведены вместе в табл.19.2.

Таблица 19.2. **Эффективность записи переменных**

Длина последовательности, байт	Число обращений к памяти	Тип описания
3	5	Целая, глобальная
4	5	Символьная, глобальная
7	8	Символьная, локальная (XCHG)
8	13	Символьная, локальная (PUSH/POP)
8	16	Целая, локальная (XCHG)
9	20	Целая, локальная (PUSH/POP)

Как вы можете видеть, целые переменные более эффективны, чем символьные, а глобальные более эффективны, чем локальные. Как это ни удивительно, но ссылки на массивы для целых требуют в два раза меньше памяти и времени исполнения, чем ссылки

на массивы символов. Так что пользуйтесь целыми, даже если нужен только один символ; вам ничего не мешает присваивать символьные значения целым переменным. Однако для больших массивов экономия памяти сводится к нулю. При описании локальных переменных вы должны стремиться целую переменную (или две целые переменные), ссылки на которую наиболее часты, описывать последней; выигрыш будет существенным.

Хотя в общем глобальные переменные более эффективны, в программах, хранящихся на диске, глобальные массивы могут занимать много места и тем самым заметно увеличивать время загрузки программ. Если же они описаны как локальные, то место на диске требуется только для команд, занимающих и освобождающих память под эти массивы, и дополнительных команд, необходимых для организации ссылок. Так что описание таких массивов как локальных может уменьшить размер программы за счет увеличения времени ее работы. Возможен и третий подход - динамическое распределение памяти. Вы можете описать глобальный указатель, а затем во время инициализации программы заказать блок памяти для массива и присвоить указателю значение адреса этого блока. Таким образом, описывая все ссылки *как глобальные и не используя массивов в самой программе*, можно получить программу минимального объема.

КОНСТАНТНЫЕ ВЫРАЖЕНИЯ В КАЧЕСТВЕ КОНСТАНТ

Так как компилятор вычисляет константные выражения в процессе компиляции, можно безнаказанно задавать константы в виде выражений, включающих несколько констант. Допустим, вы работаете с массивом символов, в котором каждые четыре байта составляют одну запись, включающую четыре однобайтовых поля. Чтобы сделать программу более ясной, вы могли бы определить идентификаторы для размера записи (например, #define SZ 4) и для смещения отдельных полей в записи (например, #define FLD3 2). Если известен указатель на какую-либо запись массива (например, aptr), то на третье поле предыдущей записи вы могли бы сослаться с помощью выражения

```
*(aptr + (FLD3 - SZ))
```

В этом случае константное выражение

```
(FLD3 - SZ)
```

генерирует команду

```
LXI H, - 2
```

так же, как если бы вы написали

```
*(aptr - 2)
```

Если бы в данном примере не было внутренних скобок, то компилятор вычислял бы это выражение так, как если бы оно было записано следующим образом:

```
*((aptr + FLD3) - SZ)
```

что далеко не так эффективно.

ПРОВЕРКА НА НУЛЬ КОРОЧЕ И БЫСТРЕЕ

Везде, где это возможно, проверяющие выражения в операторах if, for, do и while следует писать таким образом, чтобы сравнение производилось с константой, равной нулю. В этих случаях компилятор генерирует более эффективный код. Это видно из примеров в листинге 18.23. В каждом частном случае экономия зависит от того, какая операция отношения используется и какое сравнение выполняется - со знаком или без знака (адрес). При сравнении со знаком с использованием операций > и <= экономия в объеме программы небольшая, зато значительно снижается время работы за счет отсутствия обращения к библиотечным подпрограммам.

ИНДЕКСЫ В ВИДЕ НУЛЕВЫХ КОНСТАНТ НЕ СНИЖАЮТ ЭФФЕКТИВНОСТИ

Для получения адреса нужного элемента массива или указателя используются индексы. Однако если компилятор обнаруживает, что индексом является константа, равная нулю, то он совсем исключает операцию сложения при вычислении адреса. Таким образом, можно спокойно писать array[0] вместо *array. Это справедливо и в случае, когда в качестве индекса используется константное выражение. Если его вычисление дает нуль, то операция сложения пропускается.

ИСПОЛЬЗУЙТЕ ОПЕРАТОР SWITCH

Везде, где это можно, вместо строки операторов

```
if ... else ...
```

используйте оператор switch. В этом случае компилятор генерирует намного меньше команд, так как для каждого условия он генерирует только пару однословных значений (адрес и константу). Библиотечной подпрограмме CCSWITCH передается значение выражения, и она быстро пробегает по списку значений констант, ища совпадающую. При использовании же строки операторов

```
if ... else ...
```

выражение вычисляется при каждой проверке; программа в результате будет большего объема и выполняться медленнее.

СТАВЬТЕ ЗНАКИ ОПЕРАЦИЙ УВЕЛИЧЕНИЯ И УМЕНЬШЕНИЯ НА ЕДИНИЦУ ПЕРЕД ОПЕРАНДОМ

Знаки операций увеличения и уменьшения на единицу могут стоять как перед операндом, так и после него. В последнем случае результатом этих операций является исходное неизменное значение операнда, хотя фактически в процессе своей работы они изменяют операнд. Чтобы вернуть операнду его исходное значение после того, как он был увеличен или уменьшен, требуется дополнительная команда. Так, оператор

```
++ gi;
```

генерирует последовательность команд

```
LHLD GI
INX  H
SHLD GI
```

в то время как оператор

```
gi ++;
```

генерирует последовательность команд

```
LHLD GI
INX  H
SHLD GI
DCX  H
```

Часто безразлично, какое значение операнда дает операция - исходное или измененное. В подобных случаях всегда ставьте знак операции перед операндом.

ИСПОЛЬЗУЙТЕ ОПЕРАЦИИ УВЕЛИЧЕНИЯ И УМЕНЬШЕНИЯ НА ЕДИНИЦУ

Наиболее распространенные операторы, встречающиеся в программах, имеют вид

```
x = x + 1
```

Для таких операторов компилятор Смолл-Си генерирует команды

```
LHLD X
LXI  D,1
DAD  D
SHLD X
```

(где x - глобальная целая переменная), которые занимают 10 байтов и требуют 14 обращений к памяти. В противоположность этому оператор

```
++ x
```

требует только 7 байтов и 11 обращений к памяти. Это экономит 30 % памяти и на 21 % снижает количество обращений к памяти.

ИСПОЛЬЗУЙТЕ ОПЕРАЦИИ ПРИСВАИВАНИЯ ?=

Везде, где требуется выражение типа

```
x = x ? y
```

лучше написать

```
x ?= y
```

Преимущество этого состоит в том, что адрес переменной x вычисляется только один раз. Если x - простая глобальная переменная, то этот адрес не вычисляется, так как ссылка на переменную x производится прямо по имени; однако если x - элемент массива, локальный объект или же эта переменная каким-то образом вычисляется, то для получения ее адреса должны будут выполняться некоторые команды. Лучше выполнять эти команды один раз, чем дважды. Заметим, однако, что выражение

```
x += 1
```

не столь эффективно, как выражение

```
++ x
```

ИСПОЛЬЗУЙТЕ УКАЗАТЕЛИ ВМЕСТО ИНДЕКСОВ

Везде, где только можно, вы должны использовать указатели вместо индексов. Преимущество указателей состоит в том, что они прямо ссылаются на соответствующие объекты, в то время как индексированные ссылки требуют дополнительных команд для прибавления индекса (да еще, возможно, с учетом длины переменной) к адресу массива.

ИСПОЛЬЗУЙТЕ ПАРАМЕТР -O ДЛЯ УМЕНЬШЕНИЯ РАЗМЕРОВ ПРОГРАММ

В четвертой части компилятора, секции генерации кода, содержится функция peephole, которая просматривает код, сгенерированный для выражений, в том виде, в каком он записывается в выходной файл. Она ищет определенные последовательности команд на языке ассемблера и заменяет их более эффективным кодом. В некоторых случаях результатом выполняемых этой функцией преобразований является более короткая и быстродействующая программа, но в других случаях уменьшение размеров программы происходит за счет быстродействия. Одни из указанных преобразований выполняются автоматически, а другие - только по запросу пользователя.

В процессе автоматических преобразований оптимизируется выборка локальных целых переменных. Если целая переменная находится в вершине стека, то обычный код для ее выборки имеет вид


```
LXI  H,0
DAD  SP
CALL CCGINT
```

который после обработки его функцией peephole превращается в следующую последовательность команд:

```
POP  H
PUSH H
```

Для обычного кода требуется 7 байтов и 16 обращений к памяти, в то время как для последовательности команд POP и PUSH - только 2 байта и 6 обращений к памяти. В последнем случае требуется на 71 % меньше памяти и на 62 % меньше обращений к ней. Если бы потребовалась выборка во вторичный регистр (регистры DE), то в показанной выше обычной последовательности была бы необходима еще и команда XCHG. В этом случае оптимизированный код имел бы следующий вид:

```
POP  D
PUSH D
```

Если переменная является второй из двух целых переменных, находящихся в вершине стека, то она будет выбираться с помощью последовательности команд

```
POP  B
POP  H
PUSH H
PUSH B
```

или

```
POP  B
POP  D
PUSH D
PUSH B
```

в зависимости от того, какой регистр является регистром назначения - первичный или вторичный. В результате этот код требует 4 байта и 12 обращений к памяти; при этом экономится 43 % памяти и на 25 % сокращается время выполнения.

Дополнительные возможности оптимизации, выполняемой функцией peephole, связаны с заменой часто встречающихся последовательностей команд обращениями к специальным точкам входа в динамической рабочей библиотеке. Например, последовательность

```
DAD  SP
MOV  D,H
MOV  E,L
CALL CCGCHAR
INX  H
MOV  A,L
STAX D
```

заменяется на

```
CALL CCINCC
```

При этом 9 байтов и 10 обращений к памяти заменяются на 3 байта и 24 обращения к памяти, т.е. обеспечивается сокращение объема памяти на 66 % и времени выполнения на 140 %. Этот вид оптимизации задается с помощью параметра -o в командной строке при запуске компилятора.

БУДЬТЕ ВНИМАТЕЛЬНЫ ПРИ ОПРЕДЕЛЕНИИ ИМЕНИ NOCCARGC

В гл.12 было показано, как функции передается число ее аргументов. Примеры этого вы могли видеть в гл.18 при вызове функции. Определив в начале программы имя NOCCARGC, можно исключить команды загрузки в регистр ЦП числа аргументов функции. Таким образом экономятся 2 байта и 2 обращения к памяти при каждом вызове функции с аргументами и 1 байт и 1 обращение к памяти при вызове функции без аргументов. Однако вы должны быть уверены в том, что компилируемая программа не имеет функций, использующих число своих аргументов. Напомним, что функциям printf, fprintf, scanf и fscanf необходимо число аргументов, поэтому, если есть обращения к этим функциям, имя NOCCARGC определять нельзя.

ГЛАВА 20

КОМПИЛЯЦИЯ КОМПИЛЯТОРА

Две наиболее привлекательные особенности компилятора Смолл-Си заключаются в том, что он написан на своем собственном языке и пользователю предоставляется полный исходный текст компилятора. Сам компилятор является программой на языке Смолл-Си и, следовательно, может быть использован для создания новых версий самого себя. Вызов компилятора для компиляции самого себя не отличается от вызова его для компиляции других программ.

Компилятор состоит из четырех частей, каждая из которых может отдельно компилироваться и затем объединяться с остальными или при ассемблировании, или при загрузке. Если вы располагаете достаточным объемом памяти, то можно компилировать все четыре части вместе.

Исходные файлы делятся на три категории. К первой категории относится файл CC.DEF, содержащий все операторы #define для компилятора. Большинство из них служит для определения констант, а некоторые - для управления компиляцией. Определенные в этих операторах символические имена используются в командах #ifdef, #ifndef, #else и #endif для определения того, надо ли компилировать определенные строки программы. Таким образом, они определяют характеристики нового создаваемого

компилятора. Одно из этих символических имен, SEPARATE, не влияет на новый компилятор, однако если он определен, то при данном обращении к компилятору включаются только исходные файлы заданной части компилятора. Другими словами, символическое имя SEPARATE, будучи определено, сообщает компилятору о том, что вы намерены компилировать каждую часть компилятора в отдельности.

Ко второй категории исходных файлов относятся первичные файлы для каждой части компилятора: CC1.C, CC2.C, CC3.C и CC4.C. В файле CC1.C содержатся описания глобальных объектов, а в других файлах содержатся описания extern для этих объектов. Указанные описания компилируются только в том случае, если определено символическое имя SEPARATE. В каждом первичном файле есть также команда #include для головного файла STDIO.H, содержащего стандартные определения. Кроме того, в каждом первичном файле содержатся команды #include для остальных исходных файлов данной части компилятора. Файл CC1.C отличается от остальных файлов тем, что в нем содержатся также команды #include для частей 2, 3 и 4 компилятора. Но эти команды выполняются только в том случае, если не определено символическое имя SEPARATE. Из этого следует, что при компиляции компилятора целиком нужно задать только файл CC1.C. Если компиляция производится по частям, то следует задавать каждый из этих четырех файлов.

Файлы третьей категории составляют тело каждой части компилятора. В именах этих файлов содержатся две цифры. Первая цифра указывает, к какой части компилятора относится данный файл, а вторая - порядковый номер. В табл.20.1 показана взаимосвязь этих исходных файлов.

Таблица 20.1. **Исходные файлы компилятора Смолл-Си**

Часть компилятора	Первичный файл	Включаемые файлы
1	CC1.C	STDIO.H, CC.DEF, CC11.C, CC12.C, CC13.C
2	CC2.C	STDIO.H, CC.DEF, CC21.C, CC22.C
3	CC3.C	STDIO.H, CC.DEF, CC31.C, CC32.C, CC33.C
4	CC4.C	STDIO.H, CC.DEF, CC41.DEF, CC41.C, CC42.C

Ниже приводятся символические имена, предназначенные для управления характеристиками компилируемого компилятора. При отказе от функции, выполняемой компилятором, если определено какое-либо символическое имя, это символическое имя следует удалить или же оформить в виде комментария.

Символическое имя DYNAMIC служит для управления компиляцией тех операторов нового компилятора, которые динамически заказывают память под различные таблицы и буферы. Если это имя не определено, то таблицы и буферы компилируются прямо в самом компиляторе.

Задание символического имени LINK означает, что выходные файлы, которые будут создаваться новым компилятором, должны обрабатываться перемещаемым ассемблером и динамическим загрузчиком. Это имя управляет компиляцией операторов для описаний extern глобальных переменных как внешних ссылок, а всех остальных глобальных переменных как точек входа. В этом случае во время ассемблирования нельзя объединять программы, состоящие из нескольких частей, и использовать ключ задания начальных меток (ключ -b#).

Если определено символическое имя COL, то метки в выходных файлах нового компилятора будут заканчиваться двоеточием.

Символическое имя UPPER служит для задания компиляции операторов, преобразующих строчные буквы в идентификаторах в прописные; в таком виде они помещаются в таблицу имен нового компилятора. Если для конкретного ассемблера не требуется, чтобы использовались только прописные буквы, то вы можете отменить определение этого имени.

Следующие четыре символических имени позволяют указать, какие операторы языка должны поддерживаться компилятором. Чтобы компилятор был достаточно компактным для работы на компьютере с памятью емкостью 48 Кбайт, может возникнуть необходимость исключить эти операторы. Символическое имя STDO служит для управления компиляцией оператора do, STFOR - оператора for, а STSWITCH - оператора goto. Эти имена служат также для установления соответствия между каждым оператором и некоторым числовым значением; компилятор использует эти числовые значения для определения того, является ли оператор return последним оператором функции.

Символическое имя OPTIMIZE вызывает включение в новый компилятор оптимизатора peephole (в часть 4).

Все перечисленные выше символические имена, управляющие процессом компиляции, обеспечивают достаточную гибкость при адаптации компилятора к различным реализациям. Однако при всех обстоятельствах для модификации компилятора должны быть достаточно веские причины. Наиболее очевидной причиной является перенос компилятора на компьютер с ЦП, отличающимся от микропроцессора 8080. Это делается с помощью простой, но достаточно интересной процедуры.

Во-первых, следует изменить часть компилятора, генерирующую код (часть 4), так, чтобы генерируемые команды языка ассемблера воспринимались ассемблером новой машины. Затем с помощью исходного компилятора компилируется его новая версия, называемая *кросскомпилятором*. Кросскомпилятор, работая на одной машине, генерирует код для другой машины. Во время второй компиляции с помощью кросскомпилятора генерируется новый компилятор на языке ассемблера новой машины. Затем файл с новым компилятором переносится на новую машину и ассемблируется. Однако перед загрузкой и тестированием необходимо подготовить для новой машины динамическую рабочую библиотеку. Эта работа включает в себя модификацию функций библиотеки Смолл-Си с целью организации правильного их взаимодействия с операционной системой на новой машине и обработку ее кроссассемблером. Библиотека переносится на новую машину и ассемблируется. После этого программы могут переноситься на новую машину, компилироваться, ассемблироваться и выполняться.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ ТЕКСТ КОМПИЛЯТОРА СМОЛЛ-СИ

File: CC.DEF

```
/*
** Small-C Compiler Version 2.1
**
** Copyright 1982, 1983 J. E. Hendrix
**
/*
** compile options
*/
#define NOCCARGC     /* no argument counts */
#define SEPARATE      /* compile separately */
#define OPTIMIZE       /* compile output optimizer */
#define DYNAMIC       /* allocate memory dynamically */
#define COL                              /* terminate labels with a colon */
/* #define UPPER        /* force symbols to upper case */
#define LINK                             /* will use with linking loader */

/*
** machine dependent parameters
*/
#define BPW    2         /* bytes per word */
#define LBPW   1        /* log2(BPW) */
#define SBPC   1          /* stack bytes per character */
#define ERRCODE 7     /* op sys return code */

/*
** symbol table format
*/
#define IDENT    0
#define TYPE     1
#define CLASS    2
#define OFFSET   3
#define NAME     5
#define OFFSIZE (NAME-OFFSET)
#define SYMAVG  10
#define SYMMAX  14
```



```
/*
** symbol table parameters
*/
#define NUMLOCS   25
#define STARTLOC  symtab
#define ENDLOC   (symtab+(NUMLOCS*SYMAVG))
#define NUMGLBS   200
#define STARTGLB  ENDLOC
#define ENDGLB   (ENDLOC+((NUMGLBS-1)*SYMMAX))
#define SYMTBSZ   3050  /* NUMLOCS*SYMAVG + NUMGLBS*SYMMAX */

/*
** System wide name size (for symbols)
*/
#define NAMESIZE 9
#define NAMEMAX  8

/*
** possible entries for "IDENT"
*/
#define LABEL    0
#define VARIABLE 1
#define ARRAY    2
#define POINTER  3
#define FUNCTION 4

/*
** possible entries for "TYPE"
**    low order 2 bits make type unique within length
**    high order bits give length of object
*/
/*      LABEL   0 */
#define CCHAR   (1<<2)
#define CINT    (BPW<<2)

/*
** possible entries for "CLASS"
*/
/*      LABEL     0 */
#define STATIC    1
#define AUTOMATIC 2
#define EXTERNAL  3
#define AUTOEXT   4

/*
** "switch" table
*/
```

```
#define SWSIZ   (2*BPW)
#define SWTABSZ (60*SWSIZ)

/*
** "while" statement queue
*/
#define WQTABSZ  30
#define WQSIZ     3
#define WQMAX   (wq+WQTABSZ-WQSIZ)

/*
** entry offsets in while queue
*/
#define WQSP    0
#define WQLOOP  1
#define WQEXIT  2

/*
** literal pool
*/
#define LITABSZ 800
#define LITMAX  (LITABSZ-1)

/*
** input line
*/
#define LINEMAX  127
#define LINESIZE 128

/*
** output staging buffer size
*/
#define STAGESIZE   800
#define STAGELIMIT  (STAGESIZE-1)

/*
** macro (define) pool
*/
#define MACNBR   130
#define MACNSIZE (MACNBR*(NAMESIZE+2))
#define MACNEND  (macn+MACNSIZE)
#define MACQSIZE (MACNBR*7)
#define MACMAX   (MACQSIZE-1)

/*
** statement types
*/
#define STIF      1
```



```
#define STWHILE   2
#define STRETURN  3
#define STBREAK   4
#define STCONT    5
#define STASM     6
#define STEXPR    7
#define STDO      8      /* compile "do" logic */
#define STFOR     9      /* compile "for" logic */
#define STSWITCH 10    /* compile "switch/case/default" logic */
#define STCASE   11
#define STDEF    12
#define STGOTO   13     /* compile "goto" logic */
#define STLABEL  14
```

File: CC1.C

```
/*
** Small-C Compiler Part 1
*/
#include <stdio.h>
#include "notice.h"
#include "cc.def"

/*
** miscellaneous storage
*/
char
#ifdef OPTIMIZE
  optimize,   /* optimize output of staging buffer */
#endif
  alarm,      /* audible alarm on errors? */
  monitor,    /* monitor function headers? */
  pause,      /* pause for operator on errors? */
#ifdef DYNAMIC
 *stage,      /* output staging buffer */
 *symtab,     /* symbol table */
 *litq,                        /* literal pool */
 *macn,                       /* macro name buffer */
 *macq,                       /* macro string buffer */
 *pline,      /* parsing buffer */
 *mline,      /* macro buffer */
#else
  stage[STAGESIZE],
  symtab[SYMTBSZ],
  litq[LITABSZ],
  macn[MACNSIZE],
  macq[MACQSIZE],
  pline[LINESIZE],
```

```
  mline[LINESIZE],
  swq[SWTABSZ],
#endif
 *line,                         /* points to pline or mline */
 *lptr,                         /* ptr to either */
 *glbptr,       /* ptrs to next entries */
 *locptr,       /* ptr to next local symbol */
 *stagenext,  /* next addr in stage */
 *stagelast,   /* last addr in stage */
  quote[2],    /* literal string for '"' */
 *cptr,                         /* work ptrs to any char buffer */
 *cptr2,
 *cptr3,
  msname[NAMESIZE],  /* macro symbol name array */
  ssname[NAMESIZE];   /* static symbol name array */
int
#ifdef STGOTO
  nogo,                         /* > 0 disables goto statements */
  noloc,       /* > 0 disables block locals */
#endif
  op[16],      /* function addresses of binary operators */
  op2[16],     /* same for unsigned operators */
  opindex,    /* index to matched operator */
  opsize,      /* size of operator in bytes */
  swactive,   /* true inside a switch */
  swdefault,  /* default label #, else 0 */
 *swnext,     /* address of next entry */
 *swend,      /* address of last table entry */
#ifdef DYNAMIC
 *wq,                           /* while queue */
#else
  wq[WQTABSZ],
#endif
  argcs,       /* static argc */
 *argvs,       /* static argv */
 *wqptr,      /* ptr to next entry */
  litptr,       /* ptr to next entry */
  macptr,     /* macro buffer index */
  pptr,                         /* ptr to parsing buffer */
  oper,                         /* address of binary operator function */
  ch,                           /* current character of line being scanned */
  nch,                          /* next character of line being scanned */
  declared,   /* # of local bytes declared, else -1 when done */
  iflevel,     /* #if... nest level */
  skiplevel,  /* level at which #if... skipping started */
  func1,       /* true for first function */
  nxtlab,      /* next avail label # */
  litlab,       /* label # assigned to literal pool */
```



```
  beglab,      /* beginning label -- first function */
  csp,                          /* compiler relative stk ptr */
  argstk,      /* function arg sp */
  argtop,
  ncmp,                        /* # open compound statements */
  errflag,     /* non-zero after 1st error in statement */
  eof,                          /* set non-zero on final input eof */
  input,       /* fd # for input file */
  input2,      /* fd # for "include" file */
  output,      /* fd # for output file */
  files,        /* non-zero if file list specified on cmd line */
  filearg,     /* cur file arg index */
  glbflag,     /* non-zero if internal globals */
  ctext,        /* non-zero to intermix c-source */
  ccode,       /* non-zero while parsing c-code */
                                /* zero when passing assembly code */
  listfp,       /* file pointer to list device */
  lastst,       /* last executed statement type */
 *iptr;                         /* work ptr to any int buffer */

#include "cc11.c"
#include "cc12.c"
#include "cc13.c"

#ifndef SEPARATE
#include "cc21.c"
#include "cc22.c"
#include "cc31.c"
#include "cc32.c"
#include "cc33.c"
#include "cc41.c"
#include "cc42.c"
#endif
```

File: CC11.C

```
/*
** execution begins here
*/
main(argc, argv) int argc, *argv; {
  argcs=argc;
  argvs=argv;
  fputs("Small-C Compiler, ", stderr); fputs(VERSION, stderr);
  fputs(CRIGHT1, stderr);
#ifdef DYNAMIC
  swnext=calloc(SWTABSZ, 1);
  swend=swnext+((SWTABSZ-SWSIZ)>>1);
  stage=calloc(STAGESIZE, 1);
```

```
  stagelast=stage+STAGELIMIT;
  wq=calloc(WQTABSZ, BPW);
  litq=calloc(LITABSZ, 1);
  macn=calloc(MACNSIZE, 1);
  macq=calloc(MACQSIZE, 1);
  pline=calloc(LINESIZE, 1);
  mline=calloc(LINESIZE, 1);
#else
  swend=(swnext=swq)+SWTABSZ-SWSIZ;
  stagelast=stage+STAGELIMIT;
#endif
  swactive=                 /* not in switch */
  stagenext=                /* direct output mode */
  iflevel=                   /* #if... nesting level = 0 */
  skiplevel=                /* #if... not encountered */
  macptr=                   /* clear the macro pool */
  csp=                                  /* stack ptr (relative) */
  errflag=                  /* not skipping errors till ";" */
  eof=                                  /* not eof yet */
  ncmp=                                /* not in compound statement */
  files=
  filearg=
  quote[1]=0;
  func1=                    /* first function */
  ccode=1;                  /* enable preprocessing */
  wqptr=wq;                /* clear while queue */
  quote[0]='"';                         /* fake a quote literal */
  input=input2=EOF;
  ask();                     /* get user options */
  openfile();               /* and initial input file */
  preprocess();                         /* fetch first line */
#ifdef DYNAMIC
  symtab=calloc((NUMLOCS*SYMAVG + NUMGLBS*SYMMAX), 1);
#endif
  locptr=STARTLOC;
  glbptr=STARTGLB;
  glbflag=1;
  ctext=0;
  header();                  /* intro code */
  setops();                   /* set values in op arrays */
  parse();                    /* process ALL input */
  outside();                 /* verify outside any function */
  trailer();                  /* follow-up code */
  fclose(output);
  }

/*
** process all input text
```



```
**
** At this level, only static declarations,
**      defines, includes and function
**      definitions are legal...
*/
parse() {
  while (eof==0) {
    if(amatch("extern", 6))   dodeclare(EXTERNAL);
    else if(dodeclare(STATIC));
    else if(match("#asm"))    doasm();
    else if(match("#include"))doinclude();
    else if(match("#define")) addmac();
    else                      newfunc();
    blanks();                  /* force eof if pending */
    }
  }

/*
** dump the literal pool
*/
dumplits(size) int size; {
  int j, k;
  k=0;
  while (k<litptr) {
    poll(1);                    /* allow program interruption */
    defstorage(size);
    j=10;
    while(j--) {
      outdec(getint(litq+k, size));
      k=k+size;
      if ((j==0)|(k>=litptr)) {nl(); break;}
      outbyte(',');
      }
    }
  }

/*
** dump zeroes for default initial values
*/
dumpzero(size, count) int size, count; {
  int j;
  while (count > 0) {
    poll(1);                    /* allow program interruption */
    defstorage(size);
    j=30;
    while(j--) {
      outdec(0);
      if ((--count <= 0)|(j==0)) {nl(); break;}
```

```
      outbyte(',');
      }
    }
  }

/*
** verify compile ends outside any function
*/
outside()  {
  if (ncmp) error("no closing bracket");
  }

/*
** get run options
*/
ask() {
  int i;
  i=listfp=nxtlab=0;
  output=stdout;
#ifdef OPTIMIZE
  optimize=
#endif
  alarm=monitor=pause=NO;
  line=mline;
  while(getarg(++i, line, LINESIZE, argcs, argvs)!=EOF) {
    if(line[0]!='-') continue;
    if((toupper(line[1])=='L')&(isdigit(line[2]))&(line[3]<=' ')) {
      listfp=line[2]-'0';
      continue;
      }
    if(line[2]<=' ') {
      if(toupper(line[1])=='A') {
        alarm=YES;
        continue;
        }
      if(toupper(line[1])=='M') {
        monitor=YES;
        continue;
        }
#ifdef OPTIMIZE
      if(toupper(line[1])=='O') {
        optimize=YES;
        continue;
        }
#endif
      if(toupper(line[1])=='P') {
        pause=YES;
        continue;
```



```
        }
      }
#ifndef LINK
    if(toupper(line[1])=='B') {
      bump(0); bump(2);
      if(number(&nxtlab)) continue;
      }
#endif
    sout("usage: cc [file]... [-m] [-a] [-p] [-l#]", stderr);
#ifdef OPTIMIZE
    sout(" [-o]", stderr);
#endif
#ifndef LINK
    sout(" [-b#]", stderr);
#endif
    sout(NEWLINE, stderr);
    abort(ERRCODE);
    }
  }

/*
** input and output file opens
*/
openfile() {                    /* entire function revised */
  char outfn[15];
  int i, j, ext;
  input=EOF;
  while(getarg(++filearg, pline, LINESIZE, argcs, argvs)!=EOF) {
    if(pline[0]=='-') continue;
    ext = NO;
    i = -1;
    j = 0;
    while(pline[++i]) {
      if(pline[i] == '.') {ext = YES; break;}
      if(j < 10) outfn[j++] = pline[i];
      }
    if(!ext) {
      strcpy(pline + i, ".C");
      }
    input = mustopen(pline, "r");
    if(!files && isatty(stdout)) {
      strcpy(outfn + j, ".MAC");
      output = mustopen(outfn, "w");
      }
    files=YES;
    kill();
    return;
    }
```

```
  if(files++) eof=YES;
  else input=stdin;
  kill();
  }

/*
** open a file with error checking
*/
mustopen(fn, mode) char *fn, *mode; {
  int fd;
  if(fd = fopen(fn, mode)) return fd;
  sout("open error on ", stderr);
  lout(fn, stderr);
  abort(ERRCODE);
  }

setops() {
  op2[ 0]=     op[ 0]=  ffor;                    /* heir5 */
  op2[ 1]=     op[ 1]= ffxor;                    /* heir6 */
  op2[ 2]=     op[ 2]= ffand;                   /* heir7 */
  op2[ 3]=     op[ 3]=  ffeq;                    /* heir8 */
  op2[ 4]=     op[ 4]=  ffne;
  op2[ 5]=ule; op[ 5]=  ffle;                    /* heir9 */
  op2[ 6]=uge; op[ 6]=  ffge;
  op2[ 7]=ult; op[ 7]=  fflt;
  op2[ 8]=ugt; op[ 8]=  ffgt;
  op2[ 9]=     op[ 9]= ffasr;                    /* heir10 */
  op2[10]=     op[10]= ffasl;
  op2[11]=     op[11]= ffadd;                   /* heir11 */
  op2[12]=     op[12]= ffsub;
  op2[13]=     op[13]=ffmult;                   /* heir12 */
  op2[14]=     op[14]= ffdiv;
  op2[15]=     op[15]= ffmod;
  }
```

File: CC12.C

```
/*
** open an include file
*/
doinclude()  {
  char *cp;
  blanks();                                    /* skip over to name */
  switch (*lptr) {
    case '"': case '<': cp = ++lptr;
    while(*cp) {
      switch(*BQPbAyP': case '>': *cp=NULL;)
      ++cp;
      }
    }
  if((input2=fopen(lptr,"r"))==NULL) {
    input2=EOF;
    error("open failure on include file");
    }
  kill();           /* clear rest of line */
                    /* so next read will come from */
                    /* new file (if open) */
  }

/*
** test for global declarations
*/
dodeclare(class) int class; {
  if(amatch("char",4)) {
    declglb(CCHAR, class);
    ns();
    return 1;
    }
  else if((amatch("int",3))|(class==EXTERNAL)) {
    declglb(CINT, class);
    ns();
    return 1;
    }
  return 0;
  }

/*
** delcare a static variable
*/
declglb(type, class)  int type, class; {
  int k, j;
  while(1) {
    if(endst()) return;       /* do line */
    if(match("(*")|match("*")) {
      j=POINTER;
      k=0;
      }
    else {
      j=VARIABLE;
      k=1;
      }
    if (symname(ssname, YES)==0) illname();
    if(findglb(ssname)) multidef(ssname);
    if(match(")")) ;
    if(match("()")) j=FUNCTION;
```

```
    else if (match("[")) {
      paerror(j);
      k=needsub();                /* get size */
      j=ARRAY;                    /* !0=array */
      }
    if(class==EXTERNAL) external(ssname);
    else if(j!=FUNCTION) j=initials(type>>2, j, k);
    addsym(ssname, j, type, k, &glbptr, class);
    if (match(",")==0) return;    /* more? */
    }
  }

/*
** declare local variables
*/
decl1oc(typ)  int typ;  {
  int k,j;
  if(swactive) error("not allowed in switch");
#ifdef STGOTO
  if(noloc) error("not allowed with goto");
#endif
  if(declared < 0) error("must declare first in block");
  while(1) {
    while(1) {
      if(endst()) return;
      if(match("*")) j=POINTER;
      else           j=VARIABLE;
      if (symname(ssname, YES)==0) illname();
      /* no multidef check, block-locals are together */
      k=BPW;
      if (match("[")) {
        paerror(j);
        if(k=needsub()) {
          j=ARRAY;
          if(typ==CINT)k=k<<LBPW;
          }
        else {j=POINTER; k=BPW;}
        }
      else if((typ==CCHAR)&(j==VARIABLE)) k=SBPC;
      declared = declared + k;
      addsym(ssname, j, typ, csp - declared, &locptr, AUTOMATIC);
      break;
      }
    if (match(",")==0) return;
    }
  }

/*
```



```
** test for pointer array (unsupported)
*/
paerror(j) int j; {
  if(j==POINTER) error("no pointer arrays");
  }

/*
** initialize global objects
*/
initials(size, ident, dim) int size, ident, dim; {
  int savedim;
  litptr=0;
  if(dim==0) dim = -1;
  savedim=dim;
  entry();
  if(match("=")) {
    if(match("{")) {
      while(dim) {
        init(size, ident, &dim);
        if(match(",")==0) break;
        }
      needtoken("}");
      }
    else init(size, ident, &dim);
    }
  if((dim == -1)&(dim==savedim)) {
     stowlit(0, size=BPW);
    ident=POINTER;
    }
  dumplits(size);
  dumpzero(size, dim);
  return ident;
  }

/*
** evaluate one initializer
*/
init(size, ident, dim) int size, ident, *dim; {
  int value;
  if(qstr(&value)) {
    if((ident==VARIABLE)|(size!=1))
      error("must assign to char pointer or array");
    *dim = *dim - (litptr - value);
    if(ident==POINTER) point();
    }
  else if(constexpr(&value)) {
    if(ident==POINTER) error("cannot assign to pointer");
    stowlit(value, size);
```

```
    *dim = *dim - 1;
    }
  }

/*
** get required array size
*/
needsub()  {
  int val;
  if(match("]")) return 0;  /* null size */
  if (constexpr(&val)==0) val=1;
  if (val<0) {
    error("negative size illegal");
    val = -val;
    }
  needtoken("]");                          /* force single dimension */
  return val;                               /* and return size */
  }

/*
** begin a function
**
** called from "parse" and tries to make a function
** out of the following text
**
*/
newfunc()  {
  char *ptr;
#ifdef STGOTO
  nogo  =                                  /* enable goto statements */
  noloc = 0;                               /* enable block-local declarations */
#endif
  lastst=                                    /* no statement yet */
  litptr=0;                                  /* clear lit pool */
  litlab=getlabel();                        /* label next lit pool */
  locptr=STARTLOC;                      /* clear local variables */
  if(monitor) lout(line, stderr);
  if (symname(ssname, YES)==0) {
    error("illegal function or declaration");
    kill();                                   /* invalidate line */
    return;
    }
  if(func1) {
    postlabel(beglab);
    func1=0;
    }
  if(ptr=findglb(ssname)) {              /* already in symbol table ? */
    if(ptr[IDENT]!=FUNCTION)       multidef(ssname);
```



```
    else if(ptr[OFFSET]==FUNCTION) multidef(ssname);
    else {
      /* earlier assumed to be a function */
      ptr[OFFSET]=FUNCTION;
      ptr[CLASS]=STATIC;
      }
    }
  else
    addsym(ssname, FUNCTION, CINT, FUNCTION, &glbptr, STATIC);
  if(match("(")==0) error("no open paren");
  entry();
  locptr=STARTLOC;
  argstk=0;                                        /* init arg count */
  while(match(")")==0) {  /* then count args */
    /* any legal name bumps arg count */
    if(symname(ssname, YES)) {
      if(findloc(ssname)) multidef(ssname);
      else {
        addsym(ssname, 0, 0, argstk, &locptr, AUTOMATIC);
        argstk=argstk+BPW;
        }
      }
    else {error("illegal argument name");junk();}
    blanks();
    /* if not closing paren, should be comma */
    if(streq(lptr,")")==0) {
      if(match(",")==0) error("no comma");
      }
    if(endst()) break;
    }
  csp=0;                                          /* preset stack ptr */
  argtop=argstk;
  while(argstk) {
    /* now let user declare what types of things */
    /*      those arguments were */
    if(amatch("char",4))     {doargs(CCHAR);ns();}
    else if(amatch("int",3)) {doargs(CINT);ns();}
    else {error("wrong number of arguments");break;}
    }
  statement();
#ifdef STGOTO
  if(lastst != STRETURN && lastst != STGOTO) ffret();
#else
  if(lastst != STRETURN) ffret();
#endif
  if(litptr) {
    printlabel(litlab);
    col();
```

```
    dumplits(1);                        /* dump literals */
    }
  }

/*
** declare argument types
**
** called from "newfunc" this routine adds an entry in the
** local symbol table for each named argument
*/
doargs(t) int t; {
  int j, legalname;
  char c, *argptr;
  while(1) {
    if(argstk==0) return;  /* no arguments */
    if(match("(*")|match("*")) j=POINTER;  else j=VARIABLE;
    if((legalname=symname(ssname, YES))==0) illname();
    if(match(")")) ;
    if(match("()")) ;
    if(match("[")) {
      paerror(j);
      while(inbyte()!=']') if(endst()) break;          /* skip "[...]" */
      j=POINTER;                        /* add entry as pointer */
      }
    if(legalname) {
      if(argptr=findloc(ssname)) {
        /* add details of type and address */
        argptr[IDENT]=j;
        argptr[TYPE]=t;
        putint(argtop-getint(argptr+OFFSET, OFFSIZE), argptr+OFFSET, OFFSIZE);
        }
      else error("not an argument");
      }
    argstk=argstk-BPW;                  /* cnt down */
    if(endst())return;
    if(match(",")==0) error("no comma");
    }
  }
```

File: CC13.C

```
/*
** statement parser
**
** called whenever syntax requires a statement
**  this routine performs that statement
**  and returns a number telling which one
*/
statement() {
  if ((ch==0) & (eof)) return;
  else if(amatch("char",4))  {declloc(CCHAR);ns();}
  else if(amatch("int",3))   {declloc(CINT);ns();}
  else {
    if(declared >= 0) {
#ifdef STGOTO
      if(ncmp > 1) nogo=declared;       /* disable goto if any */
#endif
      csp=modstk(csp - declared, NO);
      declared = -1;
      }
    if(match("{"))                compound();
    else if(amatch("if",2))       {doif();laststst=STIF;}
    else if(amatch("while",5))    {dowhile();laststst=STWHILE;}
#ifdef STDO
    else if(amatch("do",2))       {dodo();laststst=STDO;}
#endif
#ifdef STFOR
    else if(amatch("for",3))      {dofor();laststst=STFOR;}
#endif
#ifdef STSWITCH
    else if(amatch("switch",6))   {doswitch();laststst=STSWITCH;}
    else if(amatch("case",4))     {docase();laststst=STCASE;}
    else if(amatch("default",7))  {dodefault();laststst=STDEF;}
#endif
#ifdef STGOTO
    else if(amatch("goto", 4))    {dogoto();laststst=STGOTO;}
    else if(dolabel())            ;
#endif
    else if(amatch("return",6))   {doreturn();ns();laststst=STRETURN;}
    else if(amatch("break",5))    {dobreak();ns();laststst=STBREAK;}
    else if(amatch("continue",8)) {docont();ns();laststst=STCONT;}
    else if(match(";"))           errflag=0;
    else if(match("#asm"))        {doasm();laststst=STASM;}
    else                          {doexpr();ns();laststst=STEXPR;}
    }
  return laststst;
  }

/*
** semicolon enforcer
**
** called whenever syntax requires a semicolon
*/
ns() {
  if(match(";")==0) error("no semicolon");
```

```
  else errflag=0;
  }

compound()  {
  int savcsp;
  char *savloc;
  savcsp=csp;
  savloc=locptr;
  declared=0;      /* may now declare local variables */
  ++ncmp;          /* new level open */
  while (match(")")==0)
    if(eof) {
      error("no final )");
      break;
      }
    else statement();      /* do one */
  --ncmp;                  /* close current level */
/*55*/
#ifdef STGOTO
  if(laststst != STRETURN && laststst != STGOTO)
#else
  if(laststst != STRETURN)
#endif
    modstk(savcsp, NO);      /* delete local variable space */
  csp=savcsp;
/*55*/
#ifdef STGOTO
  cptr=savloc;               /* retain labels */
  while(cptr < locptr) {
    cptr2=nextsym(cptr);
    if(cptr[IDENT] == LABEL) {
      while(cptr < cptr2) *savloc++ = *cptr++;
      }
    else cptr=cptr2;
    }
#endif
  locptr=savloc;             /* delete local symbols */
  declared = -1;             /* may not declare variables */
  }

doif()  {
  int flab1,flab2;
  flab1=getlabel();          /* get label for false branch */
  test(flab1, YES);          /* get expression, and branch false */
  statement();               /* if true, do a statement */
  if (amatch("else",4)==0) {      /* if...else ? */
    /* simple "if"...print false label */
    postlabel(flab1);
```



```
    return;                  /* and exit */
    }
  flab2=getlabel();
#ifdef STGOTO
  if((laststst != STRETURN)&(laststst != STGOTO)) jump(flab2);
#else
  if(laststst != STRETURN) jump(flab2);
#endif
  postlabel(flab1);          /* print false label */
  statement();               /* and do "else" clause */
  postlabel(flab2);          /* print true label */
  }

doexpr() {
  int const, val;
  char *before, *start;
  while(1) {
    setstage(&before, &start);
    expression(&const, &val);
    clearstage(before, start);
    if(ch != ',') break;
    bump(1);
    }
  }

dowhile()  {
  int wq[4];                 /* allocate local queue */
  addwhile(wq);              /* add entry to queue for "break" */
  postlabel(wq[WQLOOP]);     /* loop label */
  test(wq[WQEXIT], YES);     /* see if true */
  statement();               /* if so, do a statement */
  jump(wq[WQLOOP]);          /* loop to label */
  postlabel(wq[WQEXIT]);     /* exit label */
  delwhile();                /* delete queue entry */
  }

#ifdef STDO
dodo() {
  int wq[4], top;
  addwhile(wq);
  postlabel(top=getlabel());
  statement();
  needtoken("while");
  postlabel(wq[WQLOOP]);
  test(wq[WQEXIT], YES);
  jump(top);
  postlabel(wq[WQEXIT]);
  delwhile();
```

```
  ns();
  }
#endif

#ifdef STFOR
dofor() {
  int wq[4], lab1, lab2;
  addwhile(wq);
  lab1=getlabel();
  lab2=getlabel();
  needtoken("(");
  if(match(";")==0) {
    doexpr();                          /* expr 1 */
    ns();
    }
  postlabel(lab1);
  if(match(";")==0) {
    test(wq[WQEXIT], NO);              /* expr 2 */
    ns();
    }
  jump(lab2);
  postlabel(wq[WQLOOP]);
  if(match(")")==0) {
    doexpr();                          /* expr 3 */
    needtoken(")");
    }
  jump(lab1);
  postlabel(lab2);
  statement();
  jump(wq[WQLOOP]);
  postlabel(wq[WQEXIT]);
  delwhile();
  }
#endif

#ifdef STSWITCH
doswitch() {
  int wq[4], endlab, swact, swdef, *swnex, *swptr;
  swact=swactive;
  swdef=swdefault;
  swnex=swptr=swnext;
  addwhile(wq);
  *(wqptr + WQLOOP - WQSIZ) = 0;
  needtoken("(");
  doexpr();                            /* evaluate switch expression */
  needtoken(")");
  swdefault=0;
  swactive=1;
```



```
  jump(endlab=getlabel());
  statement();                                          /* cases, etc. */
  jump(wq[WQEXIT]);
  postlabel(endlab);
  sw();                                                 /* match cases */
  while(swptr < swnext) {
    defstorage(CINT>>2);
    printlabel(*swptr++);  /* case label */
    outbyte(',');
    outdec(*swptr++);                    /* case value */
    nl();
    }
  defstorage(CINT>>2);
  outdec(0);
  nl();
  if(swdefault) jump(swdefault);
  postlabel(wq[WQEXIT]);
  delwhile();
  swnext=swnex;
  swdefault=swdef;
  swactive=swact;
  }

docase() {
  if(swactive==0) error("not in switch");
  if(swnext > swend) {
    error("too many cases");
    return;
    }
  postlabel(*swnext++ = getlabel());
  constexpr(swnext++);
  needtoken(":");
  }

dodefault() {
  if(swactive) {
    if(swdefault) error("multiple defaults");
    }
  else error("not in switch");
  needtoken(":");
  postlabel(swdefault=getlabel());
  }
#endif

#ifdef STGOTO
dogoto() {
  if(nogo > 0) error("not allowed with block-locals");
  else noloc = 1;
```

```
  if(symname(ssname, YES)) jump(addlabel());
  else error("bad label");
  ns();
  }

dolabel() {
  char *savelptr;
  blanks();
  savelptr=lptr;
  if(symname(ssname, YES)) {
    if(gch()==':') {
      postlabel(addlabel());
      return 1;
      }
    else bump(savelptr-lptr);
    }
  return 0;
  }

addlabel() {
  if(cptr=findloc(ssname)) {
    if(cptr[IDENT]!=LABEL) error("not a label");
    }
  else cptr=addsym(ssname, LABEL, LABEL, getlabel(), &locptr, LABEL);
  return (getint(cptr+OFFSET, OFFSIZE));
  }
#endif

doreturn() {
  if(endst()==0) {
    doexpr();
    modstk(0, YES);
    }
  else modstk(0, NO);
  ffret();
  }

dobreak() {
  int *ptr;
  if ((ptr=readwhile(wqptr))==0) return;
  modstk((ptr[WQSP]), NO);
  jump(ptr[WQEXIT]);
  }

docont() {
  int *ptr;
  ptr = wqptr;
  while (1) {
```



```
    if ((ptr=readwhile(ptr))==0) return;
    if (ptr[WQLOOP]) break;
    }
  modstk((ptr[WQSP]), NO);
  jump(ptr[WQLOOP]);
  }

doasm() {
  ccode=0;                                  /* mark mode as "asm" */
  while (1) {
    inline();
    if (match("#endasm")) break;
    if(eof)break;
    sout(line, output);
    }
  kill();
  ccode=1;
  }
```

File: CC2.C

```
/*
** Small-C Compiler Part 2
*/
#include <stdio.h>
#include "cc.def"

extern char
#ifdef DYNAMIC
 *symtab,
 *stage,
 *macn,
 *macq,
 *pline,
 *mline,
#else
  symtab[SYMTBSZ],
  stage[STAGESIZE],
  macn[MACNSIZE],
  macq[MACQSIZE],
  pline[LINESIZE],
  mline[LINESIZE],
#endif
#ifdef OPTIMIZE
  optimize,
#endif
  alarm, *glbptr, *line, *lptr, *cptr, *cptr2,  *cptr3,
```

```
 *locptr, msname[NAMESIZE], pause, quote[2],
 *stagelast, *stagenext;
extern int
#ifdef DYNAMIC
  *wq,
#else
  wq[WQTABSZ],
#endif
  ccode, ch, csp, eof, errflag, iflevel,
  input, input2, listfp, macptr, nch,
  nxtlab, op[16], opindex, opsize, output, pptr,
  skiplevel, *wqptr;

#include "cc21.c"
#include "cc22.c"
```

File: CC21.C

```
junk() {
  if(an(inbyte())) while(an(ch)) gch();
  else while(an(ch)==0) {
    if(ch==0) break;
    gch();
    }
  blanks();
  }

endst() {
  blanks();
  return ((streq(lptr,";")|(ch==0)));
  }

illname() {
  error("illegal symbol");
  junk();
  }

multidef(sname)  char *sname; {
  error("already defined");
  }

needtoken(str)  char *str; {
  if (match(str)==0) error("missing token");
  }

needlval() {
  error("must be lvalue");
```



```
findglb(sname)  char *sname; {
  if(search(sname, STARTGLB, SYMMAX, ENDGLB, NUMGLBS, NAME))
    return cptr;
  return 0;
  }

findloc(sname)  char *sname;  {
  cptr = locptr - 1;                          /* search backward for block locals */
  while(cptr > STARTLOC) {
    cptr = cptr - *cptr;
    if(astreq(sname, cptr, NAMEMAX)) return (cptr - NAME);
    cptr = cptr - NAME - 1;
    }
  return 0;
  }

addsym(sname, id, typ, value, lgptrptr, class)
  char *sname, id, typ;  int value, *lgptrptr, class; {
  if(lgptrptr == &glbptr) {
    if(cptr2=findglb(sname)) return cptr2;
    if(cptr==0) {
      error("global symbol table overflow");
      return 0;
      }
    }
  else {
    if(locptr > (ENDLOC-SYMMAX)) {
      error("local symbol table overflow");
      abort(ERRCODE);
      }
    cptr = *lgptrptr;
    }
  cptr[IDENT]=id;
  cptr[TYPE]=typ;
  cptr[CLASS]=class;
  putint(value, cptr+OFFSET, OFFSIZE);
  cptr3 = cptr2 = cptr + NAME;
  while(an(*sname)) *cptr2++ = *sname++;
  if(lgptrptr == &locptr) {
    *cptr2 = cptr2 - cptr3; /* set length */
    *lgptrptr = ++cptr2;
    }
  return cptr;
  }

nextsym(entry) char *entry; {
```

```
  entry = entry + NAME;
  while(*entry++ >= ' '); /* find length byte */
  return entry;
  }

/*
** get integer of length len from address addr
** (byte sequence set by "putint")
*/
getint(addr, len) char *addr; int len; {
  int i;
  i = *(addr + --len);                    /* high order byte sign extended */
  while(len--) i = (i << 8) | *(addr+len)&255;
  return i;
  }

/*
** put integer i of length len into address addr
** (low byte first)
*/
putint(i, addr, len) char *addr; int i, len; {
  while(len--) {
    *addr++ = i;
    i = i>>8;
    }
  }

/*
** test if next input string is legal symbol name
*/
symname(sname, ucase) char *sname; int ucase; {
  int k;char c;
  blanks();
  if(alpha(ch)==0) return (*sname=0);
  k=0;
  while(an(ch)) {
#ifdef UPPER
    if(ucase)
      sname[k]=toupper(gch());
    else
#endif
      sname[k]=gch();
    if(k<NAMEMAX) ++k;
    }
  sname[k]=0;
  return 1;
  }
```



```
/*
** return next avail internal label number
*/
getlabel() {
  return(++nxtlab);
  }

/*
** post a label in the program
*/
postlabel(label) int label; {
  printlabel(label);
  col();
  nl();
  }

/*
** print specified number as a label
*/
printlabel(label)  int label; {
  outstr("CC");
  outdec(label);
  }

/*
** test if c is alphabetic
*/
alpha(c)  char c; {
  return (isalpha(c) || c=='_');
  }

/*
** test if given character is alphanumeric
*/
an(c)  char c; {
  return (alpha(c) || isdigit(c));
  }

addwhile(ptr)  int ptr[]; {
  int k;
  ptr[WQSP]=csp;                          /* and stk ptr */
  ptr[WQLOOP]=getlabel();                 /* and looping label */
  ptr[WQEXIT]=getlabel();                 /* and exit label */
  if (wqptr==WQMAX) {
    error("too many active loops");
    abort(ERRCODE);
    }
  k=0;
```

```
    while (k<WQSIZ) *wqptr++ = ptr[k++];
    }

  delwhile() {
    if (wqptr > wq) wqptr=wqptr-WQSIZ;
    }

  readwhile(ptr) int *ptr; {
    if (ptr <= wq) {
      error("out of context");
      return 0;
      }
    else return (ptr-WQSIZ);
  }

 white() {
 #ifdef DYNAMIC
   /* test for stack/prog overlap at deepest nesting */
   /* primary -> symname -> blanks -> white */
   avail(YES);                                  /* abort on stack overflow */
 #endif
   return (*lptr<= ' ' && *lptr!=NULL);
   }

 gch() {
   int c;
   if(c=ch) bump(1);
   return c;
   }

 bump(n) int n; {
   if(n) lptr=lptr+n;
   else  lptr=line;
   if(ch=nch = *lptr) nch = *(lptr+1);
   }

kill() {
  *line=0;
  bump(0);
  }

inbyte()  {
  while(ch==0) {
    if (eof) return 0;
    preprocess();
    }
 return gch();
 }
```



```
 inline() {
   int k,unit;                                  /* numerous revisions */
   poll(1);
   if (input==EOF) openfile();                  /* allow operator interruption */
   if(eof) return;
   if((unit=input2)==EOF) unit=input;
   if(fgets(line, LINEMAX, unit)==NULL) {
     fclose(unit);
     if(input2!=EOF) input2=EOF;
     else input=EOF;
     *line=NULL;
     }
   else if(listfp) {
     if(listfp==output) cout(';', output);
     sout(line, listfp);
     }
   bump(0);
   }
```

File: CC22.C

```
ifline() {
  while(1) {
    inline();
    if(eof) return;
    if(match("#ifdef")) {
      ++iflevel;
      if(skiplevel) continue;
      symname(msname, NO);
      if(search(msname, macn, NAMESIZE+2, MACNEND, MACNBR, 0)==0)
        skiplevel=iflevel;
      continue;
      }
    if(match("#ifndef")) {
      ++iflevel;
      if(skiplevel) continue;
      symname(msname, NO);
      if(search(msname, macn, NAMESIZE+2, MACNEND, MACNBR, 0))
        skiplevel=iflevel;
      continue;
      }
    if(match("#else")) {
      if(iflevel) {
        if(skiplevel==iflevel) skiplevel=0;
        else if(skiplevel==0)  skiplevel=iflevel;
        }
```

```
      else noiferr();
      continue;
      }
    if(match("#endif")) {
      if(iflevel) {
        if(skiplevel==iflevel) skiplevel=0;
        --iflevel;
        }
      else noiferr();
      continue;
      }
    if(skiplevel) continue;
    if(ch==0) continue;
    break;
    }
  }

keepch(c)  char c; {
  if(pptr<LINEMAX) line[++pptr]=c;
  }

preprocess() {
  int k;
  char c;
  if(ccode) {
    line=mline;
    ifline();
    if(eof) return;
    }
  else {
    line=pline;
    inline();
    return;
    }
  pptr = -1;
  while(ch != NEWLINE && ch) {
    if(white()) {
      keepch(' ');
      while(white()) gch();
      }
    else if(ch=='"') {
      keepch(ch);
      gch();
      while((ch!='"')|((*(lpt -1)==92)&(*(lptr-2)!=92))) {
        if(ch==0) {
          error("no quote");
          break;
          }
        keepch(gch());
        }
      gch();
      keepch('"');
      }
    else if(ch==39) {
      keepch(39);
      gch();
      while((ch!=39)|((*(lptr-1)==92)&(*(lptr-2)!=92))) {
        if(ch==0) {
          error("no apostrophe");
          break;
          }
        keepch(gch());
        }
      gch();
      keepch(39);
      }
    else if((ch=='/')&(nch=='*')) {
      bump(2);
      while(((ch=='*')&(nch=='/'))==0) {
        if(ch) bump(1);
        else {
          ifline();
          if(eof) break;
          }
        }
      bump(2);
      }
    else if(an(ch)) {
      k=0;
      while((an(ch)) & (k<NAMEMAX)) {
        msname[k++]=ch;
        gch();
        }
      msname[k]=0;
      if(search(msname, macn, NAMESIZE+2, MACNEND, MACNBR, 0)) {
        k=getint(cptr+NAMESIZE, 2);
        while(c=macq[k++]) keepch(c);
        while(an(ch)) gch();
        }
      else {
        k=0;
        while(c=msname[k++]) keepch(c);
        }
      }
    else keepch(gch());
```

```
    if(pptr>=LINEMAX) error("line too long");
    keepch(0);
    line=pline;
    bump(0);
    }

noiferr() {
  error("no matching #if...");
  errflag=0;
  }

addmac() {
  int k;
  if(symname(msname, NO)==0) {
    illname();
    kill();
    return;
    }
  k=0;
  if(search(msname, macn, NAMESIZE+2, MACNEND, MACNBR, 0)==0) {
    if(cptr2=cptr) while(*cptr2++ = msname[k++]);
    else {
      error("macro name table full");
      return;
      }
    }
  putint(macptr, cptr+NAMESIZE, 2);
  while(white()) gch();
  while(putmac(gch()));
  if(macptr>=MACMAX) {
    error("macro string queue full"); abort(ERRCODE);
    }
  }

putmac(c)  char c; {
  macq[macptr]=c;
  if(macptr<MACMAX) ++macptr;
  return c;
  }

/*
** search for symbol match
** on return cptr points to slot found or empty slot
*/
search(sname, buf, len, end, max, off)
  char *sname, *buf, *end;  int len, max, off; {
  cptr=cptr2=buf+((hash(sname)%(max-1))*len);
  while(*cptr != 0) {
    if(astreq(sname, cptr+off, NAMEMAX)) return 1;
    if((cptr=cptr+len) >= end) cptr=buf;
    if(cptr == cptr2) return (cptr=0);
    }
  return 0;
  }

hash(sname) char *sname; {
  int i, c;
  i=0;
  while(c = *sname++) i=(i<<1)+c;
  return i;
  }

setstage(before, start) int *before, *start; {
  if((*before=stagenext)==0) stagenext=stage;
  *start=stagenext;
  }

clearstage(before, start) char *before, *start; {
  *stagenext=0;
  if(stagenext=before) return;
  if(start) {
#ifdef OPTIMIZE
    peephole(start);
#else
    sout(start, output);
#endif
    }
  }

outdec(number)  int number; {
  int k,zs;
  char c, *q, *r;
  zs = 0;
  k=10000;
  if (number<0) {
    number=(-number);
    outbyte('-');
    }
  while (k>=1) {
    q=0; r=number;
    while(r >= k) {++q; r -= k;}
    c = q + '0';
    if ((c!='0')|(k==1)|(zs)) {
      zs=1;
      outbyte(c);
      }
```

```
    number-1;
    k=k/10;
    }
  }

ol(ptr)  char ptr[];  {
  ot(ptr);
  nl();
  }

ot(ptr) char ptr[]; {
  outstr(ptr);
  }

outstr(ptr) char ptr[]; {
  poll(1); /* allow program interruption */
  /* must work with symbol table names terminated by length */
  while(*ptr >= ' ') outbyte(*ptr++);
  }

outbyte(c) char c; {
  if(stagenext) {
    if(stagenext==stagelast) {
      error("staging buffer overflow");
      return 0;
      }
    else *stagenext++ = c;
    }
  else cout(c,output);
  return c;
  }

cout(c, fd) char c; int fd; {
  if(fputc(c, fd)==EOF) xout();
  }

sout(string, fd) char *string; int fd; {
  if(fputs(string, fd)==EOF) xout();
  }

lout(line, fd) char *line; int fd; {
  sout(line, fd);
  cout(NEWLINE, fd);
  }

xout() {
  fputs("output error", stderr);
  abort(ERRCODE);
```



```
  }

nl() {
  outbyte(NEWLINE);
  }

col() {
#ifdef COL
  outbyte(':');
#endif
  }

error(msg) char msg[]; {
  if(errflag) return; else errflag=1;
  lout(line, stderr);
  errout(msg, stderr);
  if(alarm) fputc(7, stderr);
  if(pause) while(fgetc(stderr)!=NEWLINE);
  if(listfp>0) errout(msg, listfp);
  }

errout(msg, fp) char msg[]; int fp; {
  int k; k=line+2;
  while(k++ <= lptr) cout(' ', fp);
  lout("/\\", fp);
  sout("**** ", fp); lout(msg, fp);
  }

streq(str1,str2)  char str1[],str2[]; {
  int k;
  k=0;
  while (str2[k]) {
    if ((str1[k])!=(str2[k])) return 0;
    ++k;
    }
  return k;
 }

astreq(str1,str2,len)  char str1[],str2[];int len; {
  int k;
  k=0;
  while (k<len) {
    if ((str1[k])!=(str2[k]))break;
    /*
    ** must detect end of symbol table names terminated by
    ** symbol length in binary
    */
    if(str1[k] < ' ') break;
```

```
      if(str2[k] < ' ') break;
      ++k;
      }
    if (an(str1[k]))return 0;
    if (an(str2[k]))return 0;
    return k;
   }

 match(lit)  char *lit; {
   int k;
   blanks();
   if (k=streq(lptr,lit)) {
     bump(k);
     return 1;
     }
   return 0;
   }

amatch(lit,len)  char *lit;int len; {
  int k;
  blanks();
  if (k=astreq(lptr,lit,len)) {
    bump(k);
    while(an(ch)) inbyte();
    return 1;
    }
 return 0;
}

nextop(list) char *list; {
  char op[4];
  opindex=0;
  blanks();
  while(1) {
    opsize=0;
    while(*list > ' ') op[opsize++] = *list++;
    op[opsize]=0;
    if(opsize=streq(lptr, op))
      if((*(lptr+opsize) != '=')&
         (*(lptr+opsize) != *(lptr+opsize-1)))
         return 1;
    if(*list) {
      ++list;
      ++opindex;
      }
    else return 0;
    }
}
```



```
    blanks() {
      while(1) {
        while(ch) {
          if(white()) gch();
          else return;
          }
        if(line==mline) return;
        preprocess();
        if(eof)break;
        }
      }
```

File: CC3.C

```
/*
** Small-C Compiler Part 3
*/
#include <stdio.h>
#include "cc.def"

extern char
#ifdef DYNAMIC
 *stage,
 *litq,
#else
  stage[STAGESIZE],
  litq[LITABSZ],
#endif
 *glbptr, *lptr,  ssname[NAMESIZE],  quote[2], *stagenext;
extern int
  ch,  csp,  litlab,  litptr,  nch,  op[16],  op2[16],
  oper,  opindex,  opsize;

#include "cc31.c"
#include "cc32.c"
#include "cc33.c"
```

File: CC31.C

```
/*
** lval[0] - symbol table address, else 0 for constant
** lval[1] - type of indirect obj to fetch, else 0 for static
** lval[2] - type of pointer or array, else 0 for all other
** lval[3] - true if constant expression
** lval[4] - value of constant expression (+ auxiliary uses)
```

```
** lval[5] - true if secondary register altered
** lval[6] - function address of highest/last binary operator
** lval[7] - stage address of "oper 0" code, else 0
*/

/*
** skim over terms adjoining || and && operators
*/
skim(opstr, testfunc, dropval, endval, hier, lval)
  char *opstr;
  int (*testfunc)(), dropval, endval, (*hier)(), lval[]; {
  int k, hits, droplab, endlab;
  hits=0;
  while(1) {
    k=plnge1(hier, lval);
    if(nextop(opstr)) {
      bump(opsize);
      if(hits==0) {
        hits=1;
        droplab=getlabel();
        }
      dropout(k, testfunc, droplab, lval);
      }
    else if(hits) {
      dropout(k, testfunc, droplab, lval);
      const(endval);
      jump(endlab=getlabel());
      postlabel(droplab);
      const(dropval);
      postlabel(endlab);
      lval[1]=lval[2]=lval[3]=lval[4]=lval[7]=0;
      return 0;
      }
    else return k;
    }
  }

/*
** test for early dropout from || or && evaluations
*/
dropout(k, testfunc, exit1, lval)
  int k, (*testfunc)(), exit1, lval[]; {
  if(k) rvalue(lval);
  else if(lval[3]) const(lval[4]);
  (*testfunc)(exit1);                          /* jumps on false */
  }

/*
** plunge to a lower level
*/
plnge(opstr, opoff, hier, lval)
  char *opstr;
  int opoff, (*hier)(), lval[]; {
  int k, lval2[8];
  k=plnge1(hier, lval);
  if(nextop(opstr)==0) return k;
  if(k) rvalue(lval);
  while(1) {
    if(nextop(opstr)) {
      bump(opsize);
      opindex=opindex+opoff;
      plnge2(op[opindex], op2[opindex], hier, lval, lval2);
      }
    else return 0;
    }
  }

/*
** unary plunge to lower level
*/
plnge1(hier, lval) int (*hier)(), lval[]; {
  char *before, *start;
  int k;
  setstage(&before, &start);
  k=(*hier)(lval);
  if(lval[3]) clearstage(before,0);          /* load constant later */
  return k;
  }

/*
** binary plunge to lower level
*/
plnge2(oper, oper2, hier, lval, lval2)
  int (*oper)(),(*oper2)(),(*hier)(),lval[],lval2[]; {
  char *before, *start;
  setstage(&before, &start);
  lval[5]=1;                    /* flag secondary register used */
  lval[7]=0;                    /* flag as not "... oper 0" syntax */
  if(lval[3]) {                 /* constant on left side not yet loaded */
    if(plnge1(hier, lval2)) rvalue(lval2);
    if(lval[4]==0) lval[7]=stagenext;
    const2(lval[4]<<dbltest(oper, lval2, lval));
    }
  else {                        /* non-constant on left side */
    push();
    if(plnge1(hier, lval2)) rvalue(lval2);
```

```
if(lval2[3]) {              /* constant on right side */
  if(lval2[4]==0) lval[7]=start;
  if(oper==ffadd) {    /* may test other commutative operators */
    csp=csp+2;
    clearstage(before, 0);
    const2(lval2[4]<<dbltest(oper, lval, lval2));
                                        /* load secondary */
    }
  else {
    const(lval2[4]<<dbltest(oper, lval, lval2));
                                        /* load primary */
    smartpop(lval2, start);
    }
  }
 else {                     /* non-constants on both sides */
  smartpop(lval2, start);
  if(dbltest(oper, lval,lval2)) doublereg();
  if(dbltest(oper, lval2,lval)) {
    swap();
    doublereg();
    if(oper==ffsub) swap();
    }
  }
 }
if(oper) {
  if(lval[3]=lval[3]&lval2[3]) {
    lval[4]=calc(lval[4], oper, lval2[4]);
    clearstage(before, 0);
    lval[5]=0;
    }
  else {
    if((lval[2]==0)&(lval2[2]==0)) {
      (*oper)();
      lval[6]=oper;      /* identify the operator */
      }
    else {
      (*oper2)();
      lval[6]=oper2;     /* identify the operator */
      }
    }
   if(oper==ffsub) {
     if((lval[2]==CINT)&(lval2[2]==CINT)) {
       swap();
       const(1);
       ffasr();          /** div by 2 **/
       }
     }
   if((oper==ffsub)|(oper==ffadd)) result(lval, lval2);
```



```
  }

calc(left, oper, right) int left, (*oper)(), right; {
      if(oper ==  ffor) return (left  |  right);
 else if(oper == ffxor) return (left  ^  right);
 else if(oper == ffand) return (left  &  right);
 else if(oper ==  ffeq) return (left  == right);
 else if(oper ==  ffne) return (left  != right);
 else if(oper ==  ffle) return (left  <= right);
 else if(oper ==  ffge) return (left  >= right);
 else if(oper ==  fflt) return (left  <  right);
 else if(oper ==  ffgt) return (left  >  right);
 else if(oper == ffasr) return (left  >> right);
 else if(oper == ffasl) return (left  << right);
 else if(oper == ffadd) return (left  +  right);
 else if(oper == ffsub) return (left  -  right);
 else if(oper ==ffmult) return (left  *  right);
 else if(oper == ffdiv) return (left  /  right);
 else if(oper == ffmod) return (left  %  right);
 else return 0;
 }

expression(const, val) int *const, *val;  {
  int lval[8];
  if(hier1(lval)) rvalue(lval);
  if(lval[3]) {
    *const=1;
    *val=lval[4];
    }
  else *const=0;
  }

hier1(lval)  int lval[];  {
  int k,lval2[8], lval3[2], oper;
  k=plnge1(hier3, lval);
  if(lval[3]) const(lval[4]);
      if(match("|="))  oper=ffor;
 else if(match("^="))  oper=ffxor;
 else if(match("&="))  oper=ffand;
 else if(match("+="))  oper=ffadd;
 else if(match("-="))  oper=ffsub;
 else if(match("*="))  oper=ffmult;
 else if(match("/="))  oper=ffdiv;
 elBQPbAqP"%="))  oper=ffmod;
 else if(match(">>=")) oper=ffasr;
 else if(match("<<=")) oper=ffasl;
 else if(match("="))   oper=0;
```

```
  else return k;
  if(k==0) {
    needlval();
    return 0;
    }
  lval3[0] = lval[0];
  lval3[1] = lval[1];
  if(lval[1]) {
    if(oper) {
      push();
      rvalue(lval);
      }
    plnge2(oper, oper, hier1, lval, lval2);
    if(oper) pop();
    }
  else {
    if(oper) {
      rvalue(lval);
      plnge2(oper, oper, hier1, lval, lval2);
      }
    else {
      if(hier1(lval2)) rvalue(lval2);
      lval[5]=lval2[5];
      }
    }
  store(lval3);
  return 0;
  }

hier3(lval)  int lval[]; {
  return skim("||", eq0, 1, 0, hier4, lval);
  }

hier4(lval)  int lval[]; {
  return skim("&&", ne0, 0, 1, hier5, lval);
  }

hier5(lval)  int lval[]; {
  return plnge("|", 0, hier6, lval);
  }

hier6(lval)  int lval[]; {
  return plnge("^", 1, hier7, lval);
  }

hier7(lval)  int lval[]; {
  return plnge("&", 2, hier8, lval);
  }
```



```
hier8(lval)  int lval[]; {
  return plnge("== !=", 3, hier9, lval);
  }

hier9(lval)  int lval[]; {
  return plnge("<= >= < >", 5, hier10, lval);
  }

hier10(lval)  int lval[]; {
  return plnge(">> <<", 9, hier11, lval);
  }

hier11(lval)  int lval[]; {
  return plnge("+ -", 11, hier12, lval);
  }

hier12(lval)  int lval[]; {
  return plnge("* / %", 13, hier13, lval);
  }
```

File: CC32.C

```
hier13(lval)  int lval[]; {
  int k;
  char *ptr;
  if(match("++")) {                                   /* ++lval */
    if(hier13(lval)==0) {
      needlval();
      return 0;
      }
    step(inc, lval);
    return 0;
    }
  else if(match("--")) {                              /* --lval */
    if(hier13(lval)==0) {
      needlval();
      return 0;
      }
    step(dec, lval);
    return 0;
    }
  else if (match("~")) {                              /* ~ */
    if(hier13(lval)) rvalue(lval);
    com();
    lval[4] = ~lval[4];
    return (lval[7]=0);
```

```
  }
else if (match("!")) {        /* ! */
  if(hier13(lval)) rvalue(lval);
  lneg();
  lval[4] = !lval[4];
  return (lval[7]=0);
  }
else if (match("-")) {        /* unary - */
  if(hier13(lval)) rvalue(lval);
  neg();
  lval[4] = -lval[4];
  return (lval[7]=0);
  }
else if(match("*")) {         /* unary * */
  if(hier13(lval)) rvalue(lval);
  if(ptr=lval[0])lval[1]=ptr[TYPE];
  else lval[1]=CINT;
  lval[2]=0;          /* flag as not pointer or array */
  lval[3]=0;          /* flag as not constant */
  lval[4]=1;          /* omit rvalue() on func call */
  lval[7]=0;
  return 1;
  }
else if(match("&")) {         /* unary & */
  if(hier13(lval)==0) {
    error("illegal address");
    return 0;
    }
  ptr=lval[0];
  lval[2]=ptr[TYPE];
  if(lval[1]) return 0;
                              /* global & non-array */
  address(ptr);
  lval[1]=ptr[TYPE];
  return 0;
  }
else {
  k=hier14(lval);
  if(match("++")) {           /* lval++ */
    if(k==0) {
      needlval();
      return 0;
      }
    step(inc, lval);
    dec(lval[2]>>2);
    return 0;
    }
  else if(match("--")) {      /* lval-- */
```



```
    if(k==0) {
      needlval();
      return 0;
      }
    step(dec, lval);
    inc(lval[2]>>2);
    return 0;
    }
  else return k;
  }
}

hier14(lval)  int *lval; {
  int k, const, val, lval2[8];
  char *ptr, *before, *start;
  k=primary(lval);
  ptr=lval[0];
  blanks();
  if((ch=='[')|(ch=='(')) {
    lval[5]=1;                /* secondary register will be used */
    while(1) {
      if(match("[")) {        /* [subscript] */
        if(ptr==0) {
          error("can't subscript");
          junk();
          needtoken("]");
          return 0;
          }
        else if(ptr[IDENT]==POINTER)rvalue(lval);
        else if(ptr[IDENT]!=ARRAY) {
          error("can't subscript");
          k=0;
          }
        setstage(&before, &start);
        lval2[3]=0;
        plnge2(0, 0, hier1, lval2, lval2); /* lval2 deadend */
        needtoken("]");
        if(lval2[3]) {
          clearstage(before, 0);
          if(lval2[4]) {
            if(ptr[TYPE]==CINT) const2(lval2[4]<<LBPW);
            else                const2(lval2[4]);
            ffadd();
            }
          }
        else {
          if(ptr[TYPE]==CINT) doublereg();
          ffadd();
```

```
        }
      lval[2]=0;
      lval[1]=ptr[TYPE];
      k=1;
      }
    else if(match("(")) {                    /* function(...) */
      if(ptr==0) callfunction(0);
      else if(ptr[IDENT]!=FUNCTION) {
        if(k && !lval[4]) rvalue(lval);
        callfunction(0);
        }
      else callfunction(ptr);
      k=lval[0]=lval[3]=lval[4]=0;
      }
    else return k;
    }
  }
 if(ptr==0) return k;
 if(ptr[IDENT]==FUNCTION) {
  address(ptr);
  lval[0]=0;
  return 0;
  }
 return k;
 }

primary(lval)  int *lval; {
 char *ptr, sname[NAMESIZE];
 int k;
 if(match("(")) {                                /* (expression,...) */
  do k=hier1(lval); while(match(","));
  needtoken(")");
  return k;
  }
 putint(0, lval, 8<<LBPW);                       /* clear lval array */
 if(symname(sname, YES)) {
  if(ptr=findloc(sname)) {
#ifdef STGOTO
    if(ptr[IDENT]==LABEL) {
      experr();
      return 0;
      }
#endif
    getloc(ptr);
    lval[0]=ptr;
    lval[1]=ptr[TYPE];
    if(ptr[IDENT]==POINTER) {
      lval[1]=CINT;
```



```
      lval[2]=ptr[TYPE];
      }
    if(ptr[IDENT]==ARRAY) {
      lval[2]=ptr[TYPE];
      return 0;
      }
    else return 1;
    }
  if(ptr=findglb(sname))
    if(ptr[IDENT]!=FUNCTION) {
      lval[0]=ptr;
      lval[1]=0;
      if(ptr[IDENT]!=ARRAY) {
        if(ptr[IDENT]==POINTER) lval[2]=ptr[TYPE];
        return 1;
        }
      address(ptr);
      lval[1]=lval[2]=ptr[TYPE];
      return 0;
      }
  ptr=addsym(sname,FUNCTION,CINT,0,&glbptr,AUTOEXT);
  lval[0]=ptr;
  lval[1]=0;
  return 0;
  }
 if(constant(lval)==0) experr();
 return 0;
 }

experr() {
 error("invalid expression");
 const(0);
 junk();
 }

callfunction(ptr)  char *ptr; {      /* symbol table entry or 0 */
 int nargs, const, val;
 nargs=0;
 blanks();                           /* already saw open paren */
 while(streq(lptr,")")==0) {
  if(endst()) break;
  if(ptr) {
    expression(&const, &val);
    push();
    }
  else {
    push();
    expression(&const, &val);
```

```
      swapstk();
      }
    nargs=nargs+BPW;                                /* count args*BPW */
    if (match(",")==0) break;
    }
  needtoken(")");
  if(streq(ptr+NAME, "CCARGC")==0) loadargc(nargs>>LBPW);
  if(ptr) ffcall(ptr+NAME);
  else callstk();
  csp=modstk(csp+nargs, YES);
  }
```

File: CC33.C

```
/*
** true if val1 -> int pointer or int array and val2 not ptr or array
*/
dbltest(oper, val1, val2) int (*oper)(), val1[], val2[]; {
  if((oper!=ffadd) && (oper!=ffsub)) return 0;
  if(val1[2]!=CINT) return 0;
  if(val2[2]) return 0;
  return 1;
  }

/*
** determine type of binary operation
*/
result(lval, lval2) int lval[], lval2[]; {
  if((lval[2]!=0)&(lval2[2]!=0)) {
    lval[2]=0;
    }
  else if(lval2[2]) {
    lval[0]=lval2[0];
    lval[1]=lval2[1];
    lval[2]=lval2[2];
    }
  }

step(oper, lval) int (*oper)(), lval[]; {
  if(lval[1]) {
    if(lval[5]) {
      push();
      rvalue(lval);
      (*oper)(lval[2]>>2);
      pop();
      store(lval);
      return;
```



```
      }
    else {
      move();
      lval[5]=1;
      }
    }
  rvalue(lval);
  (*oper)(lval[2]>>2);
  store(lval);
  }

store(lval)  int lval[]; {
  if(lval[1]) putstk(lval);
  else        putmem(lval);
  }

rvalue(lval) int lval[]; {
  if ((lval[0]!=0)&(lval[1]==0)) getmem(lval);
  else                           indirect(lval);
  }

test(label, parens)  int label, parens;  {
  int lval[8];
  char *before, *start;
  if(parens) needtoken("(");
  while(1) {
    setstage(&before, &start);
    if(hier1(lval)) rvalue(lval);
    if(match(",")) clearstage(before, start);
    else break;
    }
  if(parens) needtoken(")");
  if(lval[3]) {                                  /* constant expression */
    clearstage(before, 0);
    if(lval[4]) return;
    jump(label);
    return;
    }
  if(lval[7]) {                                   /* stage address of "oper 0" code *
    oper=lval[6];                                /* operator function address */
        if((oper==ffeq)|
           (oper==ule)) zerojump(eq0, label, lval);
    else if((oper==ffne)|
           (oper==ugt)) zerojump(ne0, label, lval);
    else if (oper==ffgt) zerojump(gt0, label, lval);
    else if (oper==ffge) zerojump(ge0, label, lval);
    else if (oper==uge)  clearstage(lval[7],0);
    else if (oper==fflt) zerojump(lt0, label, lval);
```

```
    else if (oper==ult)  zerojump(ult0, label, lval);
    else if (oper==ffle) zerojump(le0, label, lval);
    else                 testjump(label);
    }
  else testjump(label);
  clearstage(before, start);
  }

constexpr(val) int *val; {
  int const;
  char *before, *start;
  setstage(&before, &start);
  expression(&const, val);
  clearstage(before, 0);     /* scratch generated code */
  if(const==0) error("must be constant expression");
  return const;
  }

const(val) int val; {
  immed();
  outdec(val);
  nl();
  }

const2(val) int val; {
  immed2();
  outdec(val);
  nl();
  }

constant(lval)  int lval[]; {
  lval=lval+3;
  *lval=1;                                  /* assume it will be a constant */
  if (number(++lval)) immed();
  else if (pstr(lval)) immed();
  else if (qstr(lval)) {
    *(lval-1)=0;                            /* nope, it's a string address */
    immed();
    printlabel(litlab);
    outbyte('+');
    }
  else return 0;
  outdec(*lval);
  nl();
  return 1;
  }

number(val)  int val[]; {
```



```
  int k, minus;
  k=minus=0;
  while(1) {
    if(match("+")) ;
    else if(match("-")) minus=1;
    else break;
    }
  if(isdigit(ch)==0)return 0;
  while (isdigit(ch)) k=k*10+(inbyte()-'0');
  if (minus) k=(-k);
  val[0]=k;
  return 1;
  }

address(ptr) char *ptr; {
  immed();
  outstr(ptr+NAME);
  nl();
  }

pstr(val)  int val[]; {
  int k;
  k=0;
  if (match("'")==0) return 0;
  while(ch!=39)    k=(k&255)*256 + (litchar()&255);
  gch();
  val[0]=k;
  return 1;
  }

qstr(val)  int val[]; {
  char c;
  if (match(quote)==0) return 0;
  val[0]=litptr;
  while (ch!='"') {
    if(ch==0) break;
    stowlit(litchar(), 1);
    }
  gch();
  litq[litptr++]=0;
  return 1;
  }

stowlit(value, size) int value, size; {
  if((litptr+size) >= LITMAX) {
    error("literal queue overflow"); abort(ERRCODE);
    }
  putint(value, litq+litptr, size);
```

```
  litptr=litptr+size;
  }

/*
** return current literal char & bump lptr
*/
litchar() {
  int i, oct;
  if((ch!=92)|(nch==0)) return gch();
  gch();
  if(ch=='n') {gch(); return NEWLINE;}
  if(ch=='t') {gch(); return  9;} /* HT */
  if(ch=='b') {gch(); return  8;} /* BS */
  if(ch=='f') {gch(); return 12;} /* FF */
  i=3; oct=0;
  while(((i--)>0)&(ch>='0')&(ch<='7')) oct=(oct<<3)+gch()-'0';
  if(i==2) return gch(); else return oct;
  }
```

File: CC4.C

```
/*
** Small-C Compiler Part 4
*/
#include <stdio.h>
#include "cc.def"

extern char
 *macn,
 *cptr, *symtab,
#ifdef OPTIMIZE
 optimize,
#endif
 *stagenext, ssname[NAMESIZE];
extern int
 beglab,  csp, output;

#include "cc41.c"
#include "cc42.c"
```

File: CC41.C

```
/*
** print all assembler info before any code is generated
*/
header()  {
  beglab=getlabel();
```



```
  }

/*
** print any assembler stuff needed at the end
*/
trailer()  {
#ifndef LINK
  if((beglab == 1)|(beglab > 9000)) {
    /* implementation dependent trailer code goes here */
    }
#else
  char *ptr;
  cptr=STARTGLB;
  while(cptr<ENDGLB) {
    if(cptr[IDENT]==FUNCTION && cptr[CLASS]==AUTOEXT)
      external(cptr+NAME);
    cptr+=SYMMAX;
    }
#ifdef UPPER
  if((ptr=findglb("MAIN")) && (ptr[OFFSET]==FUNCTION))
#else
  if((ptr=findglb("main")) && (ptr[OFFSET]==FUNCTION))
#endif
    external("Ulink");       /* link to library functions */
#endif
  ol("END");
  }

/*
** load # args before function call
*/
loadargc(val) int val; {
  if(search("NOCCARGC", macn, NAMESIZE+2, MACNEND, MACNBR, 0)==0) {
    if(val) {
      ot("MVI A,");
      outdec(val);
      nl();
      }
    else ol("XRA A");
    }
  }

/*
** declare entry point
*/
entry() {
  outstr(ssname);
  col();
```

```
#ifdef LINK
  col();
#endif
  nl();
  }

/*
** declare external reference
*/
external(name) char *name; {
#ifdef LINK
  ot("EXT ");
  ol(name);
#endif
  }

/*
** fetch object indirect to primary register
*/
indirect(lval) int lval[]; {
  if(lval[1]==CCHAR) ffcall("CCGCHAR##");
  else               ffcall("CCGINT##");
  }

/*
** fetch a static memory cell into primary register
*/
getmem(lval)  int lval[]; {
  char *sym;
  sym=lval[0];
  if((sym[IDENT]!=POINTER)&(sym[TYPE]==CCHAR)) {
    ot("LDA ");
    outstr(sym+NAME);
    nl();
    ffcall("CCSXT##");
    }
  else {
    ot("LHLD ");
    outstr(sym+NAME);
    nl();
    }
  }

/*
** fetch addr of the specified symbol into primary register
*/
getloc(sym)  char *sym; {
  const(getint(sym+OFFSET, OFFSIZE)-csp);
```



```
  ol("DAD SP");
  }

/*
** store primary register into static cell
*/
putmem(lval)  int lval[]; {
  char *sym;
  sym=lval[0];
  if((sym[IDENT]!=POINTER)&(sym[TYPE]==CCHAR)) {
    ol("MOV A,L");
    ot("STA ");
    }
  else ot("SHLD ");
  outstr(sym+NAME);
  nl();
  }

/*
** put on the stack the type object in primary register
*/
putstk(lval) int lval[]; {
  if(lval[1]==CCHAR) {
    ol("MOV A,L");
    ol("STAX D");
    }
  else ffcall("CCPINT##");
  }

/*
** move primary register to secondary
*/
move() {
  ol("MOV D,H");
  ol("MOV E,L");
  }

/*
** swap primary and secondary registers
*/
swap() {
  ol("XCHG;;");                          /* peephole() uses trailing ";;" */
  }

/*
** partial instruction to get immediate value
** into the primary register
*/
```

```
immed() {
  ot("LXI H,");
  }

/*
** partial instruction to get immediate operand
** into secondary register
*/
immed2() {
  ot("LXI D,");
  }

/*
** push primary register onto stack
*/
push() {
  ol("PUSH H");
  csp=csp-BPW;
  }

/*
** unpush or pop as required
*/
smartpop(lval, start) int lval[]; char *start; {
  if(lval[5])  pop();                          /* secondary was used */
  else unpush(start);
  }

/*
** replace a push with a swap
*/
unpush(dest) char *dest; {
  int i;
  char *sour;
  sour="XCHG;;";                          /* peephole() uses trailing ";;" */
  while(*sour) *dest++ = *sour++;
  sour=stagenext;
  while(--sour > dest) {    /* adjust stack references */
    if(streq(sour,"DAD SP")) {
      --sour;
      i=BPW;
      while(isdigit(*(--sour))) {
        if((*sour = *sour-i) < '0') {
          *sour = *sour+10;
          i=1;
          }
        else i=0;
        }
```



```
      }
    }
  csp=csp+BPW;
  }

/*
** pop stack to the secondary register
*/
pop() {
  ol("POP D");
  csp=csp+BPW;
  }

/*
** swap primary register and stack
*/
swapstk() {
  ol("XTHL");
  }

/*
** process switch statement
*/
sw() {
  ffcall("CCSWITCH##");
  }

/*
** call specified subroutine name
*/
ffcall(sname)  char *sname; {
  ot("CALL ");
  outstr(sname);
  nl();
  }

/*
** return from subroutine
*/
ffret() {
  ol("RET");
  }

/*
** perform subroutine call to value on stack
*/
callstk() {
  ffcall("CCDCAL##");
```

```
  }

/*
** jump to internal label number
*/
jump(label)  int label; {
  ot("JMP ");
  printlabel(label);
  nl();
  }

/*
** test primary register and jump if false
*/
testjump(label)  int label; {
  ol("MOV A,H");
  ol("ORA L");
  ot("JZ ");
  printlabel(label);
  nl();
  }

/*
** test primary register against zero and jump if false
*/
zerojump(oper, label, lval) int (*oper)(), label, lval[]; {
  clearstage(lval[7], 0);     /* purge conventional code */
  (*oper)(label);
  }

/*
** define storage according to size
*/
defstorage(size) int size; {
  if(size==1) ot("DB ");
  else        ot("DW ");
  }

/*
** point to following object(s)
*/
point() {
  ol("DW $+2");
  }

/*
** modify stack pointer to value given
*/
```



```
modstk(newsp, save)  int newsp, save; {
  int k;
  k=newsp-csp;
  if(k==0)return newsp;
  if(k>=0) {
    if(k<7) {
      if(k&1) {
        ol("INX SP");
        k--;
        }
      while(k) {
        ol("POP B");
        k=k-BPW;
        }
      return newsp;
      }
    }
  if(k<0) {
    if(k>-7) {
      if(k&1) {
        ol("DCX SP");
        k++;
        }
      while(k) {
        ol("PUSH B");
        k=k+BPW;
        }
      return newsp;
      }
    }
  if(save) swap();
  const(k);
  ol("DAD SP");
  ol("SPHL");
  if(save) swap();
  return newsp;
  }

/*
** double primary register
*/
doublereg() {ol("DAD H");}


   File: CC42.C

/*
** add primary and secondary registers (result in primary)
```

```
*/
ffadd() {ol("DAD D");}

/*
** subtract primary from secondary register (result in primary)
*/
ffsub() {ffcall("CCSUB##");}

/*
** multiply primary and secondary registers (result in primary)
*/
ffmult() {ffcall("CCMULT##");}

/*
** divide secondary by primary register
** (quotient in primary, remainder in secondary)
*/
ffdiv() {ffcall("CCDIV##");}

/*
** remainder of secondary/primary
** (remainder in primary, quotient in secondary)
*/
ffmod() {ffdiv();swap();}

/*
** inclusive "or" primary and secondary registers
** (result in primary)
*/
ffor() {ffcall("CCOR##");}

/*
** exclusive "or" the primary and secondary registers
** (result in primary)
*/
ffxor() {ffcall("CCXOR##");}

/*
** "and" primary and secondary registers
** (result in primary)
*/
ffand() {ffcall("CCAND##");}

/*
** logical negation of primary register
*/
lneg() {ffcall("CCLNEG##");}
```



```
/*
** arithmetic shift right secondary register
** number of bits given in primary register
** (result in primary)
*/
ffasr() {ffcall("CCASR##");}

/*
** arithmetic shift left secondary register
** number of bits given in primary register
** (result in primary)
*/
ffasl() {ffcall("CCASL##");}

/*
** two's complement primary register
*/
neg() {ffcall("CCNEG##");}

/*
** one's complement primary register
*/
com() {ffcall("CCCOM##");}

/*
** increment primary register by one object of whatever size
*/
inc(n) int n; {
  while(1) {
    ol("INX H");
    if(--n < 1) break;
    }
  }

/*
** decrement primary register by one object of whatever size
*/
dec(n) int n; {
  while(1) {
    ol("DCX H");
    if(--n < 1) break;
    }
  }

/*
** test for equal to
*/
ffeq()  {ffcall("CCEQ##");}
```

```
/*
** test for equal to zero
*/
eq0(label) int label; {
  ol("MOV A,H");
  ol("ORA L");
  ot("JNZ ");
  printlabel(label);
  nl();
  }

/*
** test for not equal to
*/
ffne()  {ffcall("CCNE##");}

/*
** test for not equal to zero
*/
ne0(label) int label; {
  ol("MOV A,H");
  ol("ORA L");
  ot("JZ ");
  printlabel(label);
  nl();
  }

/*
** test for less than (signed)
*/
fflt()  {ffcall("CCLT##");}

/*
** test for less than to zero
*/
lt0(label) int label; {
  ol("XRA A");
  ol("ORA H");
  ot("JP ");
  printlabel(label);
  nl();
  }

/*
** test for less than or equal to (signed)
*/
ffle()  {ffcall("CCLE##");}
```



```
/*
** test for less than or equal to zero
*/
le0(label) int label; {
  ol("MOV A,H");
  ol("ORA L");
  ol("JZ $+8");
  ol("XRA A");
  ol("ORA H");
  ot("JP ");
  printlabel(label);
  nl();
  }

/*
** test for greater than (signed)
*/
ffgt()  {ffcall("CCGT##");}

/*
** test for greater than to zero
*/
gt0(label) int label; {
  ol("XRA A");
  ol("ORA H");
  ot("JM ");
  printlabel(label);
  nl();
  ol("ORA L");
  ot("JZ ");
  printlabel(label);
  nl();
  }

/*
** test for greater than or equal to (signed)
*/
ffge()  {ffcall("CCGE##");}

/*
** test for gteater than or equal to zero
*/
ge0(label) int label; {
  ol("XRA A");
  ol("ORA H");
  ot("JM ");
  printlabel(label);
```

```
 nl();
 }

/*
** test for less than (unsigned)
*/
ult()  {ffcall("CCULT##");}

/*
** test for less than to zero (unsigned)
*/
ult0(label) int label; {
  ot("JMP ");
  printlabel(label);
  nl();
  }

 /*
 ** test for less than or equal to (unsigned)
 */
 ule()  {ffcall("CCULE##");}

 /*
 ** test for greater than (unsigned)
 */
 ugt()  {ffcall("CCUGT##");}

 /*
 ** test for greater than or equal to (unsigned)
 */
 uge()  {ffcall("CCUGE##");}

 #ifdef OPTIMIZE
 peephole(ptr) char *ptr; {
   while(*ptr) {
     if(streq(ptr,"LXI H,0\nDAD SP\nCALL CCGINT##")) {
       if(streq(ptr+29, "XCHG;;")) {pp2();ptr=ptr+36;}
       else                        {pp1();ptr=ptr+29;}
       }
     else if(streq(ptr,"LXI H,2\nDAD SP\nCALL CCGINT##")) {
       if(streq(ptr+29, "XCHG;;")) {pp3(pp2);ptr=ptr+36;}
       else                        {pp3(pp1);ptr=ptr+29;}
       }
     else if(optimize) {
       if(streq(ptr, "DAD SP\nCALL CCGINT##")) {
         ol("CALL CCDSGI##");
         ptr=ptr+21;
         }
     else if(streq(ptr, "DAD D\nCALL CCGINT##")) {
       ol("CALL CCDDGI##");
       ptr=ptr+20;
       }
     else if(streq(ptr, "DAD SP\nCALL CCGCHAR##")) {
       ol("CALL CCDSGC##");
       ptr=ptr+22;
         }
     else if(streq(ptr, "DAD D\nCALL CCGCHAR##")) {
       ol("CALL CCDDGC##");
       ptr=ptr+21;
       }
     else if(streq(ptr,
       "DAD SP\nMOV D,H\nMOV E,L\nCALL CCGINT##\nINX H\nCALL CCPINT##")) {
       ol("CALL CCINCI##");
       ptr=ptr+57;
       }
     else if(streq(ptr,
       "DAD SP\nMOV D,H\nMOV E,L\nCALL CCGINT##\nDCX H\nCALL CCPINT##")) {
       ol("CALL CCDECI##");
       ptr=ptr+57;
       }
     else if(streq(ptr,
       "DAD SP\nMOV D,H\nMOV E,L\nCALL CCGCHAR##\nINX H\nMOV A,L\nSTAX D")) {
       ol("CALL CCINCC##");
       ptr=ptr+59;
       }
     else if(streq(ptr,
       "DAD SP\nMOV D,H\nMOV E,L\nCALL CCGCHAR##\nDCX H\nMOV A,L\nSTAX D")) {
       ol("CALL CCDECC##");
       ptr=ptr+59;
       }
     else if(streq(ptr, "DAD D\nPOP D\nCALL CCPINT##")) {
       ol("CALL CDPDPI##");
       ptr=ptr+26;
       }
     else if(streq(ptr, "DAD D\nPOP D\nMOV A,L\nSTAX D")) {
       ol("CALL CDPDPC##");
       ptr=ptr+27;
       }
     else if(streq(ptr, "POP D\nCALL CCPINT##")) {
       ol("CALL CCPDPI##");
       ptr=ptr+20;
       }
     /* additional optimizing logic goes here */
     else cout(*ptr++, output);
     }
   else cout(*ptr++, output);
```

```
    }
  }

pp1() {
  ol("POP H");
  ol("PUSH H");
  }

pp2() {
  ol("POP D");
  ol("PUSH D");
  }

pp3(pp) int (*pp)(); {
  ol("POP B");
  (*pp)();
  ol("PUSH B");
  }
#endif
```

ПРИЛОЖЕНИЕ Б

БИБЛИОТЕКА АРИФМЕТИЧЕСКИХ И ЛОГИЧЕСКИХ ПОДПРОГРАММ

File: CALL.MAC

```
;
;----- CALL: Small-C arithmetic and logical library
;
CCDCAL::
        PCHL
;
CCDDGC::
        DAD     D
        JMP     CCGCHAR
;
CCDSGC::
        INX     H
        INX     H
        DAD     SP
;
;FETCH A SINGLE BYTE FROM THE ADDRESS IN HL AND SIGN INTO HL
CCGCHAR::
        MOV     A,M
;
;PUT THE ACCUM INTO HL AND SIGN EXTEND THROUGH H.
CCARGC::
CCSXT::
        MOV     L,A
        RLC
        SBB     A
        MOV     H,A
        RET
;
CCDDGI::
        DAD     D
        JMP     CCGINT
;
CCDSGI::
        INX     H
        INX     H
        DAD     SP
;
;FETCH A FULL 16-BIT INTEGER FROM THE ADDRESS IN HL INTO HL
CCGINT::
        MOV     A,M
        INX     H
        MOV     H,M
        MOV     L,A
        RET
;
CCDECC::
        INX     H
        INX     H
        DAD     SP
        MOV     D,H
        MOV     E,L
        CALL    CCGCHAR
        DCX     H
        MOV     A,L
        STAX    D
        RET
;
CCINCC::
        INX     H
        INX     H
        DAD     SP
        MOV     D,H
        MOV     E,L
        CALL    CCGCHAR
```

```
        INX     H
        MOV     A,L
        STAX    D
        RET
;
CDPDPC::
        DAD     D
CCPDPC::
        POP     B       ;RET ADDR
        POP     D
        PUSH    B
;
;STORE A SINGLE BYTE FROM HL AT THE ADDRESS IN DE
CCPCHAR::
PCHAR:  MOV     A,L
        STAX    D
        RET
;
CCDECI::
        INX     H
        INX     H
        DAD     SP
        MOV     D,H
        MOV     E,L
        CALL    CCGINT
        DCX     H
        JMP     CCPINT
;
CCINCI::
        INX     H
        INX     H
        DAD     SP
        MOV     D,H
        MOV     E,L
        CALL    CCGINT
        INX     H
        JMP     CCPINT
;
CDPDPI::
        DAD     D
CCPDPI::
        POP     B       ;RET ADDR
        POP     D
        PUSH    B
;
;STORE A 16-BIT INTEGER IN HL AT THE ADDRESS IN DE
CCPINT::
PINT:   MOV     A,L
```



```
        STAX    D
        INX     D
        MOV     A,H
        STAX    D
        RET
;
;INCLUSIVE "OR" HL AND DE INTO HL
CCOR::
        MOV     A,L
        ORA     E
        MOV     L,A
        MOV     A,H
        ORA     D
        MOV     H,A
        RET
;
;EXCLUSIVE "OR" HL AND DE INTO HL
CCXOR::
        MOV     A,L
        XRA     E
        MOV     L,A
        MOV     A,H
        XRA     D
        MOV     H,A
        RET
;
;"AND" HL AND DE INTO HL
CCAND::
        MOV     A,L
        ANA     E
        MOV     L,A
        MOV     A,H
        ANA     D
        MOV     H,A
        RET
;
;IN ALL THE FOLLOWING COMPARE ROUTINES, HL IS SET TO 1 IF THE
;  CONDITION IS TRUE, OTHERWISE IT IS SET TO 0 (ZERO).
;
;TEST IF HL = DE
;
CCEQ::
        CALL    CCCMP
        RZ
        DCX     H
        RET
;
;TEST IF DE != HL
```

```
CCNE::
        CALL    CCCMP
        RNZ
        DCX     H
        RET
;
;TEST IF DE > HL (SIGNED)
CCGT::
        XCHG
        CALL    CCCMP
        RC
        DCX     H
        RET
;
;TEST IF DE <= HL (SIGNED)
CCLE::
        CALL    CCCMP
        RZ
        RC
        DCX     H
        RET
;
;TEST IF DE >= HL (SIGNED)
CCGE::
        CALL    CCCMP
        RNC
        DCX     H
        RET
;
;TEST IF DE < HL (SIGNED)
CCLT::
        CALL    CCCMP
        RC
        DCX     H
        RET
;
;COMMON ROUTINE TO PERFORM A SIGNED COMPARE OF DE AND HL
; THIS ROUTINE PERFORMS DE - HL AND SETS THE CONDITIONS:
; CARRY REFLECTS SIGN OF DIFFERENCE (SET MEANS DE < HL)
; ZERO/NON-ZERO SET ACCORDING TO EQUALITY.
CCCMP::
        MOV     A,H     ;INVERT SIGN OF HL
        XRI     80H
        MOV     H,A
        MOV     A,D     ;INVERT SIGN OF DE
        XRI     80H
        CMP     H       ;COMPARE MSBS
        JNZ     CCCMP1  ;DONE IF NEQ
```



```
        MOV     A,E     ;COMPARE LSBS
        CMP     L
CCCMP1: LXI H,1         ;PRESET TRUE COND
        RET
;
;TEST IF DE >= HL (UNSIGNED)
CCUGE::
        CALL    CCUCMP
        RNC
        DCX     H
        RET
;
;TEST IF DE < HL (UNSIGNED)
CCULT::
        CALL    CCUCMP
        RC
        DCX     H
        RET
;
;TEST IF DE > HL (UNSIGNED)
CCUGT::
        XCHG
        CALL    CCUCMP
        RC
        DCX     H
        RET
;
;TEST IF DE <= HL (UNSIGNED)
CCULE::
        CALL    CCUCMP
        RZ
        RC
        DCX     H
        RET
;
;COMMON ROUTINE TO PERFORM UNSIGNED COMPARE
; CARRY SET IF DE < HL
; ZERO/NONZERO SET ACCORDINGLY
CCUCMP::
        MOV     A,D
        CMP     H
        JNZ     UCMP1
        MOV     A,E
        CMP     L
UCMP1:  LXI     H,1
        RET
;
;SHIFT DE ARITHMETICALLY RIGHT BY HL AND RETURN IN HL
```

```
CCASR::
        XCHG
        DCR     E
        RM
        MOV     A,H
        RAL
        MOV     A,H
        RAR
        MOV     H,A
        MOV     A,L
        RAR
        MOV     L,A
        JMP     CCASR+1
;
;SHIFT DE ARITHMETICALLY LEFT BY HL AND RETURN IN HL
CCASL::
        XCHG
        DCR     E
        RM
        DAD     H
        JMP     CCASL+1
;
;SUBTRACT HL FROM DE AND RETURN IN HL
CCSUB::
        MOV     A,E
        SUB     L
        MOV     L,A
        MOV     A,D
        SBB     H
        MOV     H,A
        RET
;
;FORM THE TWO'S COMPLEMENT OF HL
CCNEG::
        CALL    CCCOM
        INX     H
        RET
;
;FORM THE ONE'S COMPLEMENT OF HL
CCCOM::
        MOV     A,H
        CMA
        MOV     H,A
        MOV     A,L
        CMA
        MOV     L,A
        RET
;
```



```
;MULTIPLY DE BY HL AND RETURN IN HL (SIGNED MULTIPLY)
CCMULT::
MULT:   MOV     B,H
        MOV     C,L
        LXI     H,0
MULT1:  MOV     A,C
        RRC
        JNC     MULT2
        DAD     D
MULT2:  XRA A
        MOV     A,B
        RAR
        MOV     B,A
        MOV     A,C
        RAR
        MOV     C,A
        ORA     B
        RZ
        XRA     A
        MOV     A,E
        RAL
        MOV     E,A
        MOV     A,D
        RAL
        MOV     D,A
        ORA     E
        RZ
        JMP     MULT1
;
;DIVIDE DE BY HL AND RETURN QUOTIENT IN HL, REMAINDER IN DE (SIGNED DIVIDE)
CCDIV::
DIV:    MOV     B,H
        MOV     C,L
        MOV     A,D
        XRA     B
        PUSH    PSW
        MOV     A,D
        ORA     A
        CM      CCDENEG
        MOV     A,B
        ORA     A
        CM      CCBCNEG
        MVI     A,16
        PUSH    PSW
        XCHG
        LXI     D,0
CCDIV1: DAD     H
        CALL    CCRDEL
```

```
        JZ      CCDIV2
        CALL    CCCMPBCDE
        JM      CCDIV2
        MOV     A,L
        ORI     1
        MOV     L,A
        MOV     A,E
        SUB     C
        MOV     E,A
        MOV     A,D
        SBB     B
        MOV     D,A
CCDIV2: POP     PSW
        DCR     A
        JZ      CCDIV3
        PUSH    PSW
        JMP     CCDIV1
CCDIV3: POP     PSW
        RP
        CALL    CCDENEG
        XCHG
        CALL    CCDENEG
        XCHG
        RET
;
;NEGATE THE INTEGER IN DE (INTERNAL ROUTINE)
CCDENEG: MOV    A,D
        CMA
        MOV     D,A
        MOV     A,E
        CMA
        MOV     E,A
        INX     D
        RET
;
;NEGATE THE INTEGER IN BC (INTERNAL ROUTINE)
CCBCNEG: MOV    A,B
        CMA
        MOV     B,A
        MOV     A,C
        CMA
        MOV     C,A
        INX     B
        RET
;
;ROTATE DE LEFT ONE BIT (INTERNAL ROUTINE)
CCRDEL: MOV     A,E
        RAL
```



```
        MOV     E,A
        MOV     A,D
        RAL
        MOV     D,A
        ORA     E
        RET
;
;COMPARE BC TO DE (INTERNAL ROUTINE)
CCCMPBCDE: MOV  A,E
        SUB     C
        MOV     A,D
        SBB     B
        RET
;
;LOGICAL NEGATION
CCLNEG::
        MOV     A,H
        ORA     L
        JNZ     $+6
        MVI     L,1
        RET
        LXI     H,0
        RET
;
; EXECUTE "SWITCH" STATEMENT
;
;  HL  =  SWITCH VALUE
; (SP) -> SWITCH TABLE
;         DW ADDR1, VALUE1
;         DW ADDR2, VALUE2
;         ...
;         DW 0
;        [JMP default]
;         continuation
;
CCSWITCH::
        XCHG            ;DE =  SWITCH VALUE
        POP     H       ;HL -> SWITCH TABLE
SWLOOP: MOV     C,M
        INX     H
        MOV     B,M     ;BC -> CASE ADDR, ELSE 0
        INX     H
        MOV     A,B
        ORA     C
        JZ      SWEND   ;DEFAULT OR CONTINUATION CODE
        MOV     A,M
        INX     H
        CMP     E
```

```
        MOV     A,M
        INX     H
        JNZ     SWLOOP
        CMP     D
        JNZ     SWLOOP
        MOV     H,B     ;CASE MATCHED
        MOV     L,C
SWEND:  PCHL
;
        END
```

ПРИЛОЖЕНИЕ В

СОВМЕСТИМОСТЬ С ПОЛНОЙ ВЕРСИЕЙ ЯЗЫКА СИ

В то время как в большинстве реализаций полной версии языка Си в качестве признака новой строки используется символ "перевод строки", в Смолл-Си могут применяться как символ "возврат каретки", так и символ "перевод строки". Следовательно, для перехода на новую строку всегда задавайте последовательность \n, а не числовую константу (с. 47).

При вычислении выражений компилятор Смолл-Си все неописанные имена интерпретирует как имена функций. В полной версии языка Си так считается только в том случае, если имя задано как вызов функции (т.е. если далее следует открывающая круглая скобка). Таким образом, необходимо избегать ссылок на имена функций, за исключением вызовов функций (с. 62, 68).

Компилятор Смолл-Си допускает описание формального параметра, являющегося именем функции, в виде int arg, и arg(...) для вызова этой функции. В полной версии языка Си с той же целью необходимо задавать описания int (*arg)() и (*arg)(...) соответственно. Для обеспечения совместимости с другими компиляторами следует придерживаться синтаксиса полной версии языка Си (с. 63).

В Смолл-Си любое выражение, за которым следует открывающая круглая скобка, является вызовом функции, однако в полной версии языка Си требуется, чтобы имя функции или выражение, как правило, выглядели так: (*func). Любые вольности здесь могут создать несовместимость с компиляторами полной версии языка Си (с. 62).

В Смолл-Си целые константы всегда считаются десятичными, в то врем как в полной версии языка Си нуль в начале числа воспринимается как признак восьмеричной константы. Следовательно, чтобы избежать ошибочной трактовки констант в полной версии языка Си, лучше избегать задания ведущих нулей (с. 46).

Смолл-Си в последовательности цифр со служебным символом для восьмеричного числа допускает только цифры от 0 до 7, в то время как полная версия языка Си допускает также цифры 8 и 9, которые воспринимаются как восьмеричные числа 10 и 11 (с. 47).

В Смолл-Си при выборке символа из памяти происходит распространение знака. Это выполняется не во всех компиляторах полной версии языка Си, и такое различие в операциях выборки может стать причиной несовместимости (с. 48).

В Смолл-Си вызываемой функции указывается, сколько ей было передано аргументов. В полной версии языка Си этого нет. Использовать такую возможность следует в крайних случаях (с. 64), поскольку это приводит к несовместимости с полной версией языка Си.

В Смолл-Си, в отличие от полной версии языка Си, допускается задание нескольких префиксов case с одним и тем же значением. Это следует иметь в виду (с. 80).

В Смолл-Си, в отличие от полной версии языка Си, можно не заключать имя файла в команде #include в кавычки или угловые скобки. Для обеспечения совместимости с компилятором ОС UNIX можно задать для стандартного файла заголовка #include <stdio.h>, а для остальных файлов - #include "*имя файла*". Однако более старые версии Смолл-Си не воспринимают этих разделителей (с. 87).

Компилятор Смолл-Си не распознает принятые в ранних версиях операции присваивания с ведущим знаком равенства. Поэтому последовательности типа =*, =& и т. п. воспринимаются как пары знаков операций. В полной версии языка Си, однако, они воспринимаются как знаки операций присваивания *=, &= и т.д. Поэтому во избежание двусмысленности при написании последовательности вроде =* ставьте пробел между знаками операций (с. 73).

В Смолл-Си левая часть в операции присваивания вычисляется до вычисления правой части. Это значит, что переменные (например, индексы), которым присваиваются значения, не изменяются при вычислении присваиваемых значений. Однако многие компиляторы полной версии языка Си сначала обрабатывают правую часть, позволяя, таким образом, определить получающий значение объект. Поэтому следует избегать таких присваиваний, когда переменные левой части изменяются в правой части оператора присваивания (с. 74).

Компилятор Смолл-Си вычисляет значения выражений, являющихся фактическими аргументами, в том порядке, в каком они перечислены. Поэтому возможно влияние таких аргументов на значения аргументов, находящиеся справа от них. В полной

версии языка Си порядок вычисления значений аргументов не определен, поэтому лучше не использовать списки аргументов, в которых порядок вычисления может влиять на передаваемые значения (с. 63).

В Смолл-Си для идентификации файлов в функциях ввода-вывода используются описатели файлов, в то время как в стандартных функциях ОС UNIX используются указатели. Однако это различие не вызывает несовместимости, так как значения, передаваемые функциям ввода-вывода, являются или значениями, возвращаемыми функцией fopen, или одним из стандартных символических имен stdin, stdout или stderr (с. 96).

Спецификации формата b для функций printf и fscanf воспринимаются только компилятором Смолл-Си. Следовательно, их использование может повлиять на возможность переноса программ в другую среду (с. 101, 104).

ПРИЛОЖЕНИЕ Г

СООБЩЕНИЯ ОБ ОШИБКАХ

Когда компилятор обнаруживает ошибку, он выводит на консоль строку, в которой произошла ошибка. Под этой строкой, приблизительно под местом ошибки, выводится стрелка, образованная символами /\, за которой следует сообщение об ошибке. Если в командной строке задан ключ -p, то компилятор после выдачи сообщения об ошибке делает паузу и ждет, пока оператор не нажмет клавишу возврата каретки (RETURN), вызывающую продолжение работы.

Для некоторых программ внутренний буфер компилятора может переполняться. Из этой ситуации возможны два выхода: перекомпилировать компилятор, увеличив размер переполняемого буфера, или изменить программу с целью исключения возможности переполнения. Последний вариант может потребоваться в том случае, если объем памяти не достаточен для работы с буфером большего размера.

Ниже в алфавитном порядке перечислены выдаваемые компилятором сообщения об ошибках и даны объяснения этих сообщений с указанием предполагаемых причин возникновения ошибок. Иногда сообщение об ошибке не отражает действительных причин ее возникновения, так как эта ошибка не обрабатывается компилятором. Поэтому текст сообщения об ошибке не всегда точно соответствует ситуации.

already defined (уже определено)

На глобальном уровне или среди формальных параметров функции одно и то же имя определяется более одного раза.

bad label (неправильная метка)

Ошибка в метке в операторе goto. Метка не отвечает соглашениям, принятым в языке Си по образованию имен, или же метка совсем отсутствует.

can't subscript (индекс не допускается)

Индекс относится к идентификатору, который не соответствует указателю или массиву.

cannot assign to pointer (не может быть присвоено указателю)

Начальное значение, содержащее константу или константное выражение, присваивается указателю. Инициализация указателей целых переменных не допускается; для указателей символьных переменных допускается использование в качестве начальных значений только списка выражений или строки, содержащей константы.

global symbol table overflow (переполнена таблица глобальных символов)

Произошло переполнение таблицы глобальных символов. Для исправления этой ошибки можно перекомпилировать компилятор, присвоив константам NUMGLBS и SYMTBSZ (определенным в файле CC.DEF) большие значения. NUMGLBS - число записей в таблице глобальных имен, а SYMTBSZ - общий размер таблиц локальных и глобальных имен в байтах. Порядок вычисления значения SYMTBSZ объясняется в комментарии к исходному тексту.

illegal address (недопустимый адрес)

Операнд операции получения адреса не является переменной, указателем (с индексом или без индекса) или именем массива.

illegal argument name (недопустимое имя аргумента)

Имя формального параметра в описании аргументов функции не отвечает соглашениям по образованию имен, принятым в языке Си.

illegal function or declaration (недопустимая функция или описание)

После обработки строки препроцессором компилятор обнаружил на глобальном уровне элемент, не являющийся функцией или описанием.

illegal symbol (недопустимый идентификатор)

Компилятор обнаружил идентификатор, не соответствующий соглашениям по образованию имен, принятым в языке Си.

invalid expression (неправильное выражение)

Выражение содержит элемент, не являющийся константой, константной строкой или правильным именем языка Си. Заметим, что не определенные ранее имена (если они правильно сформированы) считаются именами функций, и сообщения об ошибках не выдаются.

line too long (слишком длинная строка)

После обработки препроцессором строка исходного текста имеет длину больше LINEMAX (80 символов). Эту ошибку можно исправить, разбив строку на части. Однако можно перекомпилировать компилятор, увеличив значения констант LINEMAX и LINESIZE (определенные в файле CC.DEF). Заметим, что значение константы LINESIZE должно быть на единицу больше значения константы LINEMAX.

literal queue overflow (переполнена очередь литералов)

Возможно переполнение очереди литералов, создаваемой компилятором. Очередь литералов - это буфер, в котором компилятор сохраняет строки, содержащие константы, до тех пор, пока он не достигнет конца функции. В этот момент он переписывает их в выходной файл и очищает буфер для следующей функции. Буфер литералов можно увеличить, перекомпилировав компилятор с большим значением константы LITABSZ, определенным в файле CC.DEF. LITABSZ - это размер буфера литералов в байтах. Напомним, что каждая строка, содержащая константы, заканчивается в буфере нулевым байтом.

local symbol table overflow (переполнена таблица локальных имен)

Возможно переполнение таблицы локальных имен. Таблица локальных имен - это таблица, содержащая аргументы, передаваемые функции, и локальные переменные, описанные внутри функции. После каждой функции она очищается для использования в следующей функции. Таким образом, ее размер должен быть достаточным для того, чтобы помещались имена, используемые только в одной функции. Эту ошибку можно исправить, перекомпилировав компилятор с большими значениями констант NUMLOCS и SYMTBSZ (определенными в файле CC.DEF). NUMLOCS - это число записей в таблице, а SYMTBSZ - общий размер таблиц локальных и глобальных символов в байтах. Порядок вычисления значения константы SYMTBSZ объясняется в комментариях в исходном тексте.

macro name table full (таблица макроопределений заполнена)

Команда #define может переполнить таблицу макроопределений. Эту таблицу можно расширить, перекомпилировав компилятор и увеличив значения констант MACNBR и MACNSIZE (определенные в файле CC.DEF). MACNBR - это число имен, помещающихся в таблице, а MACNSIZE - размер таблицы макроопределений в байтах. Порядок вычисления значения константы MACNSIZE объясняется в комментариях в исходном тексте.

macro string queue full (заполнена очередь строк макроопределений)

Команда #define может переполнить очередь строк макроопределений. Очередь строк макроопределений - это буфер для замены строк, связанных с именами макроопределений. Эту проблему можно разрешить, перекомпилировав компилятор с большим значением константы MACQSIZE (определенным в файле CC.DEF). MACQSIZE - это размер буфера строк макроопределений в байтах. В этот буфер должны помещаться все заменяемые строки, определенные во всей компилируемой программе. Каждая строка заканчивается нулевым символом.

missing token (отсутствует элемент синтаксиса)

Отсутствует элемент, который согласно правилам синтаксиса необходим.

multiple defaults (больше одного префикса default)

В операторе switch содержится больше одного префикса default.

mast assign to char pointer or array (должно присваиваться указателю символов или массиву)

Строка, содержащая константы, присваивается в качестве начального значения переменной, не являющейся указателем символов или массива символов.

must be constant expression (должно быть константное выражение)

На месте константного выражения, требуемого в соответствии с правилами синтаксиса, обнаружено что-то другое.

must be lvalue (должно быть lvalue)

В качестве поля назначения в выражении использовано не lvalue; lvalue - это

какое-либо выражение (возможно, просто имя), соответствующее ячейке памяти, которая может быть изменена. Это сообщение может быть выдано при попытке присвоить значение константе или имени массива без индекса.

must declare first in block (должно объявляться первым в блоке)

Локальное описание расположено после первого предложения блока.

negative size illegal (отрицательный размер недопустим)

Размерность массива задана отрицательным числом. Напомним, что в качестве размерности массива может быть использовано константное выражение. Вычисление такого выражения вполне может дать отрицательное значение.

no apostrophe (нет апострофа)

В конце символьной константы отсутствует апостроф.

no closing bracket (нет закрывающей скобки)

Входной файл компилятора заканчивается внутри тела функции.

no comma (нет запятой)

Отсутствует разделительная запятая в списке аргументов или описаний переменных.

no final } (нет закрывающей фигурной скобки)

Входной файл компилятора заканчивается внутри составного оператора.

no matching #if... (нет соответствующей команды #if...)

Для команды #else или #endif отсутствует команда #ifdef или #ifndef.

no open paren (нет открывающей круглой скобки)

При явном описании функции отсутствует открывающая круглая скобка перед списком формальных параметров.

no quote (нет кавычек)

Отсутствуют кавычки в конце строки констант. Напомним, что строка констант не может продолжаться на следующей строке исходного текста, поэтому открывающие и закрывающие кавычки должны находиться в одной и той же строке.

no semicolon (нет точки с запятой)

Отсутствует символ "точка с запятой", необходимый по правилам синтаксиса.

not a label (не метка)

Имя, следующее за ключевым словом goto, определено в программе, но не является меткой.

not allowed with block-locals (не допускается внутри блока с локальными описаниями)

Обнаружен оператор goto внутри функции, имеющей локальные описания на уровне ниже, чем заголовок функции. Это не допускается в Смолл-Си.

not allowed with goto (не допускается при goto)

Обнаружено локальное описание на уровне ниже, чем тело функции, содержащей оператор goto. Это не допускается в Смолл-Си.

not allowed with switch (не допускается в операторе switch)

Обнаружено локальное описание в теле оператора switch. Это не допускается в Смолл-Си.

not an argument (не аргумент)

Имена в списке формальных параметров в заголовке функции не соответствуют типам переменных.

not in switch (не в операторе switch)

Префиксы case или default обнаружены вне оператора switch.

open error on *имя файла* (ошибка при открытии включаемого в текст файла)

Не может быть открыт файл, заданный в команде #include.

out of context (вне контекста)

Оператор break находится вне оператора do, for, while или switch, либо оператор continue находится вне оператора do, for или while.

output error (ошибка при выводе)

Обнаружена ошибка при записи в выводимый файл. Это сообщение может быть вызвано ошибкой ввода-вывода, наличием защиты файла на диске или недостатком места на диске.

staging buffer overflow (промежуточный буфер переполнен)

Объем кода, генерируемого для выражения, превышает размер промежуточного буфера. В промежуточном буфере временно хранятся все команды, генерируемые при обработке выражения, так что в нем могут производиться различные изменения. По достижении конца выражения этот буфер переписывается в выходной файл и освобождается для следующего выражения. Выйти из создавшейся ситуации можно, разбив выражение на несколько промежуточных выражений или перекомпилировав компилятор, увеличив значение константы STAGESIZE (определенное в файле CC.DEF). STAGESIZE - это размер промежуточного буфера в байтах.

too many active loops (слишком много активных циклов)

Уровень вложенности каких-либо комбинаций операторов do, for, while и switch превышает возможности *временной очереди*, предназначенной для хранения меток начала и конца циклов. Это сообщение не относится к оператору case, так как в этом случае циклы не организуются. Данной ошибки можно избежать, увеличив значение константы WQTABSZ (определенное в файле CC.DEF). WQTABSZ - это размер временной очереди в байтах. Значение константы WQTABSZ должно быть кратно значению константы WQSZ.

too many cases (слишком много префиксов case)

Для префиксов case в операторе switch не хватает места в таблице переходов. Эту таблицу можно увеличить, увеличив значение константы SWTABSZ (определенное в файле CC.DEF) и перекомпилировав компилятор. SWTABSZ - это размер таблицы переходов в байтах. Значение константы SWTABSZ должно быть кратно значению константы SWSIZE.

wrong number of arguments (неправильное число аргументов)

До входа в тело функции не задан тип одного или более формальных аргументов, заданных в заголовке функции.

ПРИЛОЖЕНИЕ Д

НАБОР СИМВОЛОВ КОДА ASCII

Д	В	Ш			Д	В	Ш		Д	В	Ш	
0	0	0	^@	NUL	44	54	2C	,	88	130	58	X
1	1	1	^A	SOH	45	55	2D	-	89	131	59	Y
2	2	2	^B	STX	46	56	2E	.	90	132	5A	Z
3	3	3	^C	ETX	47	57	2F	/	91	133	5B	[
4	4	4	^D	EOT	48	60	30	0	92	134	5C	\
5	5	5	^E	ENQ	49	61	31	1	93	135	5D	]
6	6	6	^F	ACK	50	62	32	2	94	136	5E	
7	7	7	^G	BEL	51	63	33	3	95	137	5F	
8	10	8	^H	BS	52	64	34	4	96	140	60	`
9	11	9	^I	HT	53	65	35	5	97	141	61	a
10	12	A	^J	LF	54	66	36	6	98	142	62	b
11	13	B	^K	VT	55	67	37	7	99	143	63	c
12	14	C	^L	FF	56	70	38	8	100	144	64	d
13	15	D	^M	CR	57	71	39	9	101	145	65	e
14	16	E	^N	SO	58	72	3A	:	102	146	66	f
15	17	F	^O	SI	59	73	3B	;	103	147	67	g
16	20	10	^P	DLE	60	74	3C	<	104	150	68	h
17	21	11	^Q	DC1	61	75	3D	=	105	151	69	i
18	22	12	^R	DC2	62	76	3E	>	106	152	6A	j
19	23	13	^S	DC3	63	77	3F	?	107	153	6B	k
20	24	14	^T	DC4	64	100	40	@	108	154	6C	l
21	25	15	^U	NAK	65	101	41	A	109	155	6D	m
22	26	16	^V	SYN	66	102	42	B	110	156	6D	n
23	27	17	^W	ETB	67	103	43	C	111	157	6F	o
24	30	18	^X	CAN	68	104	44	D	112	160	70	p
25	31	19	^Y	EM	69	105	45	E	113	161	71	q
26	32	1A	^Z	SUB	70	106	46	F	114	162	72	r
27	33	1B	^[	ESC	71	107	47	G	115	163	73	s
28	34	1C	^\	FS	72	110	48	H	116	164	74	t
29	35	1D	^]	GS	73	111	49	I	117	165	75	u
30	36	1E	^^	RS	74	112	4A	J	118	166	76	v
31	37	1F	^_	US	75	113	4B	K	119	167	77	w
32	40	20		SP	76	114	4C	L	120	170	78	x
33	41	21	!		77	115	4D	M	121	171	79	y
34	42	22	"		78	116	4E	N	122	172	7A	z
35	43	23	#		79	117	4F	O	123	173	7B	{
36	44	24	$		80	120	50	P	124	174	7C	|
37	45	25	%		81	121	51	Q	125	175	7D	}
38	46	26	&		82	122	52	R	126	176	7E	~
39	47	27	'		83	123	53	S	127	177	7F	DEL
40	50	28	(		84	124	54	T				
41	51	29	)		85	125	55	U				
42	52	2A	*		86	126	56	V				
43	53	2B	+		87	127	57	W				

Д - двоичное значение, В - восьмеричное значение, Ш - шестнадцатеричное значение.

ПРИЛОЖЕНИЕ Е

СИСТЕМА КОМАНД МИКРОПРОЦЕССОРА 8080 КРАТКИЙ СПРАВОЧНИК

Символ	Значение
=	Замена
<=>	Обмен операндов
_	Логическое объединение элементов
<AND>	Поразрядная операция И
<OR>	Поразрядная операция ИЛИ
<XOR>	Поразрядная операция исключающее ИЛИ
<NOT>	Дополнение до единицы
[x]	Однобайтовый операнд в порте ввода-вывода x
(x)	Однобайтовый операнд в ячейке памяти x
(x,y)	Двухбайтовый операнд в ячейках памяти x и y (x относится к старшему по значению байту)
M	Операнд в ячейке памяти, адрес которой задан в HL
*	Флаг изменяется в соответствии с результатом
V	При переполнении условие PE, иначе PO
IE	Триггер разрешения прерывания
CY	Флаг переноса
S	Знаковый разряд
n	Однобайтовое целое без знака (в диапазоне десятичных чисел 0...255)
nn	Двухбайтовое целое без знака (в диапазоне десятичных чисел 0...65535)
d	Однобайтовое целое со знаком (в диапазоне десятичных чисел -128 ..0...+127)
p	0H, 8H, 10H, 18H, 20H, 28H, 30H или 38H
cc	C, NC, Z, NZ, PE, PO, M или P (мнемонические значения флагов)
r	A, B, C, D, E, H или L
r'	A, B, C, D, E, H или L
rm	A, B, C, D, E, H, L или M
rr	B, D, H или SP (пары регистров)
ss	B, D, H или SPW (пары регистров)

8-разрядная загрузка

Формат		Функция
LDA	nn	A=(nn)
LDAX	B	A=(BC)
LDAX	D	A=(DE)
MVI	r,n	r=n
MOV	r,r'	r=r'
MOV	r,M	r=(HL)
STA	nn	(nn)=A
STAX	B	(BC)=A
STAX	D	(DE)=A
MVI	M,n	(HL)=n
MOV	M,r	(HL)=r

Арифметика над разрядами

Формат		Функция	1=C 0=NC	Z NZ	PE PO	M P
ADI	n	A=A+n	*	*	*	*
ADD	rm	A=A+rm	*	*	*	*
ACI	n	A=A+n+CY	*	*	*	*
ADC	rm	A=A+rm+CY	*	*	*	*
SUI	n	A=A-n	*	*	*	*
SUB	rm	A=A-rm	*	*	*	*
SBI	n	A=A-n-CY	*	*	*	*
SBB	rm	A=A-rm-CY	*	*	*	*
DAA		Коррекция A	*	*	*	*
INR	rm	rm=rm+1		*	*	*
DCR	rm	rm=rm-1		*	*	*

16-разрядная загрузка

Формат		Функция
LXI	rr,nn	rr=nn
LHLD	nn	HL=(nn+1,nn)
SHLD	nn	(nn+1,nn)=HL
SPHL		SP=HL
PUSH	ss	(SP-1,SP-2)=ss SP=SP-2
POP	ss	ss=(SP+1,SP) SP=SP+2

16-разрядная арифметика

Формат		Функция	1 = C 0 = NC
DAD	rr	HL=HL+rr	*
INX	rr	rr=rr+1	
DCX	rr	rr=rr-1	

Обмен

Формат	Функция
XCHG	DE<=>HL
XTHL	HL<=>(SP+1,SP)

Циклический сдвиг

Формат	Функция	1 = C 0 = NC
RLC	A циклически сдвигается влево	*
RAL	CY_A циклически сдвигаются вправо	*
RRC	A циклически сдвигается вправо	*
RAR	A_CY циклически сдвигаются вправо	*

Логика

Формат		Функция	1 = C 0 = NC	Z NZ	PE PO	M P
ANI	n	A=<AND>N	NC	*	*	*
ANA	rm	A=A<AND>rm	NC	*	*	*
ORI	n	A=A<OR>n	NC	*	*	*
ORA	rm	A=A<OR>rm	NC	*	*	*
XRI	n	A=A<XOR>n	NC	*	*	*
XRA	rm	A=A<XOR>rm	NC	*	*	*
CPI	n	A-n	*	*	*	*
CMP	rm	A-rm	*	*	*	*
CMA		A=<NOT>A				
CMC		CY=<NOT>CY	*			
STC		CY=1	C			

Переход

Формат		Функция
JMP	nn	PC=nn
Jcc	nn	PC=nn if cc
PCHL		PC=HL

Вызов и возврат

Формат		Функция
CALL	nn	(SP-1,SP-2)=PC SP=SP-2 PC=nn
Ccc	nn	CALL if cc
RET		PC=(SP+1,SP) SP=SP+2
Rcc		RET if cc true
RST	p	(SP-1,SP-2) SP=SP-2 PC=p

Управление ЦП

Формат	Функция
NOP	Нет операции
HLT	Останов ЦП
DI	IE=0
EI	IE=1

Ввод-вывод

Формат		Функция
IN	n	A=[n]
OUT	n	[n]=A

ПРИЛОЖЕНИЕ Ж

ЯЗЫК СМОЛЛ-СИ. КРАТКИЙ СПРАВОЧНИК

Этот краткий справочник не является формальным описанием языка Смол-Си. Для сокращения его размера были опущены очевидные определения. Основные термины печатаются курсивом и могут связываться дефисами в единые синтаксические единицы. Косая черта / также служит для связи терминов. Ее следует читать как *и/или*. Символы, набранные полужирным шрифтом, являются составными частями синтаксиса языка. Под словом *строка* понимается ряд символов, написанных подряд без разделяющих пробелов. Слово *список* означает ряд элементов, разделенных запятыми и необязательными пробелами. Многоточие означает повторение подобных элементов. Апостроф в конце термина служит для обозначения необязательного элемента.

СИНТАКСИС ЯЗЫКА

Описание-аргумента:

описание-объекта

Список аргументов:

список-имен

Команда:

#include *"имя-файла"*
#include *<имя-файла>*
#include *имя-файла*
#define *имя строка-символов'*
#ifdef *имя*
#ifndef *имя*
#else
#endif
#asm
#endasm

Константа:

целое
'символ' (допускается последовательность со служебным символом)
'символ символ' (допускаются последовательности со служебным символом)

Константное выражение:

константа
знак-операции константное-выражение

константное-выражение знак-операции константное-выражение
(константное-выражение)

Описатель:

объект начальное-значение' (только инициализация глобальных объектов)

Последовательность со служебным символом:

\n (новая строка)
\t (табуляция)
\b (возврат на шаг)
\f (управление форматом)
*****восьмеричное целое*
*****символ*

Выражение:

первичное-значение
знак-операции выражение
выражение знак-операции
выражение знак-операции выражение

Описание функции:

имя (список-аргументов') описание-аргумента'...
{описание-объекта'...оператор'...}

Глобальное описание:

описание-объекта
описание-функции

Начальное значение:

= константное-выражение
= {список-константных-выражений}
= константная-строка

Имя:

строка-из-букв/цифр/подчеркивающих-линий (не начинается с цифры)

Объект:

имя
***имя**
имя [*константное-выражение'*]

Описание объекта:

тип список-описаний;
extern *тип' список-описаний;* (только для глобальных объектов)

Первичное значение:

имя
константа
константная-строка
имя[*выражение*]
первичное-значение (*список-выражений*)
(*выражение*)

Программа:

команда / *глобальное-описание...*

Оператор:

;
список-выражений;
return f *список-выражений*
имя: оператор
goto *имя*;
if (*список-выражений*) *оператор*
if (*список-выражений*) *оператор* else *оператор*
switch (*список-выражений*) {*оператор*...}
case *константное-выражение: оператор*
default: *оператор*
break;
while (*список-выражений*) *оператор*
for (*список-выражений*;
список-выражений;
список-выражений) *оператор*
do *оператор* while (*список-выражений*);
continue;
{*описание-объекта* ... *оператор* ...}

Константная строка:

"*символьная-строка*" (допускаются последовательности со служебным символом)

Тип:

char
int

СТАНДАРТНЫЕ ФУНКЦИИ

Функция	Возвращаемое значение
abs (nbr) int nbr;	Абсолютное значение
abort (errcode) int errcode;	
atoi (str) char *str;	Значение
atoib (str, base) char *str; int base;	Значение

avail (abort) int abort;	#число доступных байтов, нуль
calloc (nbr, sz) int nbr, sz;	Указатель, нуль
cfree (addr) char *addr;	addr,нуль
clearerr (fd) int fd;	
cseek (fd, offset, from) int fd, offset, from;	Нуль, EOF
ctell (fd) int fd;	Относительный номер блока
delete (name) char *name;	Нуль, ERR
dtoi (str, nbr) char *str; int *nbr	Длина поля
exit (errcode) int errcode;	
fclose (fd) int fd;	Нуль, не нуль
feof (fd) int fd;	Нуль, не нуль
ferror (fd) int fd	Нуль, не нуль
fflush (fd) int fd;	Нуль, EOF
fgetc (fd) int fd;	Символ, EOF
fgets (str, sz, fd) char *str; int sz, fd;	str, нуль
fopen (name, mode) char *name, *mode;	fd, нуль
fprintf (fd, str, arg1, arg2,...) int fd; char *str;	#число выданных символов
fputc (c,fd) char c; int fd;	c, EOF
fputs (str, fd) char *str; int fd;	
*fread (ptr, sz, cnt,fd) char *ptr; int sz, cnt, fd;*	#число прочитанных элементов
free (addr) char *addr;	addr, нуль
freopen (name, mode, fd) char *name, *mode; int fd;	fd, нуль
fscanf (fd, str, arg1, arg2,...) int fd; char *str;	#число просмотренных полей, EOF
fwrite (ptr, sz, cnt, fd) char *ptr; int sz, cnt, fd;	#число записанных элементов
getarg (nbr, str, sz, argc, argv) char *str; int nbr,sz, argc, *argv;	Длина поля, EOF
getc (fd) int fd;	Символ, EOF
getchar ()	Символ, EOF
gets (str) char *str;	str, нуль
isalnum(c) char c;	Не нуль, нуль
isalpha (c) char c;	Не нуль, нуль
isascii (c) char c;	Не нуль, нуль
isatty (fd) int fd;	Не нуль, нуль
iscntrl (c) char c;	Не нуль, нуль
iscons (fd) int fd;	Не нуль, нуль
isdigit (c) char c;	Не нуль, нуль
isgraph (c) char c;	Не нуль, нуль
islower (c) char c;	Не нуль, нуль
isprint (c) char c;	Не нуль, нуль
ispunct (c) char c;	Не нуль, нуль
isspace (c) char c;	Не нуль, нуль
isupper (c) char c;	Не нуль, нуль
isxdigit (c) char c;	Не нуль, нуль
itoa (nbr, str) int nbr; char *str;	

```
itoab (nbr, str,base) int nbr; char *str; int base;
itod (nbr, str,sz) int nbr, sz; char *str;                          str
itoo (nbr, str,sz) int nbr, sz; char *str;                          str
itou (nbr, str,sz) int nbr, sz; char *str;                          str
itox (nbr, str,sz) int nbr, sz; char *str;                          str
left (str) char *str;
lexcmp (str1, str2) char *str1, *str2;                          <0, 0,>0
lexorder (c1, c2) char c1, c2;                                  <0, 0,>0
malloc (nbr) int nbr;                                    Указатель, нуль
otoi (str, nbr) char *str; int *nbr;                          Длина поля
poll (pause) int pause;                                     Символ, нуль
printf (str, arg1, arg2,...) char *str;      Число выведенных символов
putc (c, fd) char c; int fd;                                      c, EOF
putchar (c) char c;                                               c, EOF
puts (str) char *str;
read (fd,ptr,cnt) int fd, cnt; char *ptr;     Число прочитанных байтов
reverse (str) char *str;
rewind (fd) int fd;                                          Нуль, EOF
scanf (str, arg1, arg2,...) char *str;                          Число
                                                 просмотренных полей, EOF
sign (nbr) int nbr;                                           -1, 0, +1
strcat (dest, sour) char *dest, *sour;                             dest
strchr (str, c) chr *str, c;                             Указатель, нуль
strcmp (str1, str2) char *str1, *str2;                          <0, 0,>0
strcpy (dest, sour) char *dest, *sour;                             dest
strlen (str) char *str;                                    Длина строки
strncat (dest, sour, n) char *dest, *sour, int n;                  dest
strncmp (str1, str2,n) char *str1,*str2,int n;                  <0, 0,>0
strncpy (dest, sour, n) char *dest, *sour; int n;                  dest
strrchr (str, c) char *str, c;                           Указатель, нуль
toascii (c) char c;                                 Значение в коде ASCII
tolower (c) char c;                                       Строчная буква
toupper (c) char c;                                       Прописная буква
ungetc (c, fd) char c; int fd;                                    c, EOF
unlink (name) char *name;                                    Нуль, ERR
utoi (str, nbr) char *str; int *nbr;                          Длина поля
write (fd, ptr, cnt) int fd, cnt; char *ptr;  Число записанных байтов
xtoi (str, nbr) char *str; int *nbr;                          Длина поля
```


СПИСОК ЛИТЕРАТУРЫ

Cain, Ron. "A Small C Compiler for the 8080's". *Dr. Dobb's Journal,* April-May 1980, pp.5-19.

-. "A Runtime Library for the Small C Compiler." *Dr. Dobb's Journal,* September 1980, pp.4-15.

Calingaert, Peter. *Assemblers, Compilers, and Program Translation.* Potomac, Md.,: Computer Science Press, 1979.

Hendrix, J.E. "Small-C Compiler, v.2." *Dr. Dobb's Journal,* December 1982, pp.16-53, and January 1983, pp.48-64.

Johnson, S.C.; Kernighan, B.W.; Lesk, M.E.; and Ritchie, D.M. "The C Programming Language." *The Bell System Technical Journal,* July-August 1978, pp.1991-2019.

Kernighan, Brian W., and Ritchie, Dennis M. *The C Programming Language.* Englewood Cliffs, N.J.,: Prentice-Hall, 1978. [Имеется перевод: **Керниган Б., Ритчи Д., Фьюэр А.** Язык программирования Си. Задачи по языку Си: Пер. с англ. - М.: Финансы и статистика, 1985. - 279 с.]

Miller, Alan R. *8080/Z80 Assembly Language.* New York: John Wiley & Sons, 1981.

Производственное издание

Хендрикс Джеймс

КОМПИЛЯТОР ЯЗЫКА СИ ДЛЯ МИКРОЭВМ

Заведующий редакцией А.А. Э й д е с
Редакторы О.В. Т о л к а ч е в а, М.С. Г о р д о н
Обложка художника В.Я. Ш а п о ш н и к о в а
Художественный редактор А.С. Ш и р о к о в
Технический редактор Г.З. К у з н е ц о в а

ИБ № 1735

Подписано в печать с оригинала-макета 21.08.89. Формат 60x88/16. Бумага офс. № 2 Гарнитура "Пресс-роман". Печать офсетная. Усл. печ. л. 14,70. Усл. кр.-отт. 15,07. Уч.-изд. л. 11,63. Тираж 30 000 экз. Изд. № 22272. Заказ № 1360. Цена 80 к.
Издательство "Радио и связь". 101000 Москва, Почтамт, а/я 693

Московская типография № 4 Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли. 129041 Москва, Б. Переяславская ул., д. 46