ЕВГЕНИЙ КОРНИЛОВ

bhv

ПРОГРАММИРОВАНИЕ ШАХМАТ И ДРУГИХ ЛОГИЧЕСКИХ ИГР

МЕТОДИКИ И АЛГОРИТМЫ ПРОГРАММИРОВАНИЯ ШАХМАТНЫХ ПРОГРАММ

ПРОВЕРЕННЫЕ ПРИЕМЫ ПРОГРАММИРОВАНИЯ ЛОГИЧЕСКИХ ИГР

ПРИМЕРЫ НА ЯЗЫКАХ C++ И PASCAL

PRO
ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

+CD

Евгений Корнилов

ПРОГРАММИРОВАНИЕ ШАХМАТ И ДРУГИХ ЛОГИЧЕСКИХ ИГР

Санкт-Петербург
«БХВ-Петербург»
2005

УДК 681.3.068
ББК 32.973
К67

Корнилов Е. Н.

К67 Программирование шахмат и других логических игр. — СПб.: БХВ-Петербург, 2005. — 272 с.: ил.

ISBN 5-94157-497-5

Рассмотрено программирование логических игр методом перебора на примере шахмат. Описываются стандартные методики создания шахматной программы, а также приемы, позволяющие разрабатывать более эффективные компьютерные логические игры. Представлены примеры использования рассмотренных методов при программировании других логических игр ("крестики-нолики", "уголки", шашки). Приведено большое количество исходных кодов программ на языках C++ и Pascal и полезных практических советов. На компакт-диске содержатся наиболее известные открытые коды шахматных программ, а также исходные тексты программ, написанных автором.

Для программистов

УДК 681.3.068
ББК 32.973

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Елена Кашлакова*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 09.12.04.
Формат $70\times100^{1}/_{16}$. Печать офсетная. Усл. печ. л. 21,93.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-497-5

Содержание

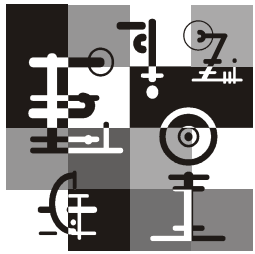
Введение

"Неприлично рассуждать о шахматном
программировании, не имея
работающей программы"
Кен Томпсон

Машина играет в шахматы — разве это не удивительно! Шахматы считаются самой интеллектуальной игрой, и вдруг в нее играет машина. Когда-то подобное казалось невероятным, но сейчас к этому привыкли. Такое впечатление, что изобретен искусственный интеллект. На самом деле, это, конечно, не так. Машина играет настолько умно, насколько ее научит человек. Есть простые игры, типа "Крестиков-ноликов", с ограниченным количеством ходов. В них машину ничему не нужно учить, она легко просчитывает ситуацию до конца и играет идеально. В шахматах количество ходов намного больше, и дерево перебора огромно. До конца просчитать ситуацию, кроме некоторых редких случаев, просто немыслимо. Как же машина играет и выбирает лучший ход? Об этом и пытается рассказать эта книга. Ее условно можно поделить на две части. В первой дается история развития шахматных программ и алгоритмов, далее описаны современные стандартные методики оптимизации перебора, приведена сравнительно несложная шахматная программа. Вторая часть рассказывает о более изощренной технике выборочного поиска и отсечении статической оценкой. Эта часть более трудна для понимания, особенно для не знакомых с этой техникой. Там нет готовых рецептов, ясны лишь направления, куда нужно двигаться. Все современные программы в той или иной мере используют эту технику. Даже сугубые консерваторы, не верящие ни в какой forward prunnig (отсечение ветвей без их предварительного рассмотрения), тем не менее, используют эту же технику в своих форсированных вариантах, которые считают только взятия. Автор надеется, что книга будет полезна всем интересующимся данной тематикой. От читателя не требуется никакой специальной подготовки, только знание какого-либо алгоритмического языка. Книга может быть полезна и более продвинутым в данной области. Они могут найти в ней некоторые идеи или просто альтернативный взгляд на

некоторые вещи. Хочется подчеркнуть, что все описываемые методики известны (некоторые уже давно) и (в той или иной мере) используются во многих программах, даже чемпионов среди компьютерных программ. Автора очень интересует обмен мнениями по данной проблеме. Замечания или просто обсуждение того или иного алгоритма, а также иной взгляд на решение затронутых проблем можно прислать по адресам: **e_k@sbor.net**, **kevgeniy@yandex.ru**.

Приятного чтения!



ГЛАВА 1

Общие сведения

История развития шахматных программ

Когда же в первый раз машина заиграла в шахматы? На этот вопрос трудно ответить, наверное, в 1769 году. Венгерский инженер барон Вольфганг фон Компелен создал машину, способную играть в шахматы (рис. 1.1). Она была предназначена для развлечения королевы Марии-Терезии. Машина, действительно, неплохо играла — внутри ее сидел сильный шахматист, который и делал ходы. История примечательная и, в некоторой степени, актуальна и в наши дни. Что программист вложит в машину, то она и играет. Гарри Каспаров, например, понимает этот тезис буквально. Он утверждает, что в его втором, решающем матче с DeepBlue (первый он выиграл), начиная со второй партии, в игру машины вмешивался человек и компенсировал неспособность машины понимать позиционную игру. Оставим этот вопрос на совести корпорации IBM и Гарри Каспарова. Я лично нисколько не сомневаюсь, что Каспаров может выиграть еще один матч с DeepBlue (если этот матч возможен), если, конечно, ему повезет.

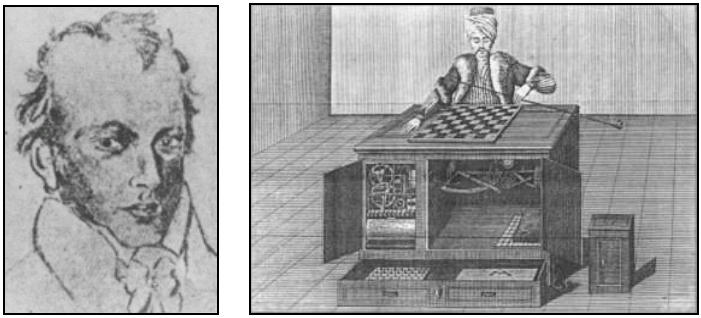


Рис. 1.1. Вольфганг фон Компелен и "играющий турка"

В 1951 году Алан Тьюринг (рис. 1.2) написал алгоритм, при помощи которого машина могла бы играть в шахматы. Только в роли машины выступал сам изобретатель. Этот нонсенс даже получил название — "бумажная машина Тьюринга". Автору требовалось более получаса, чтобы сделать один ход. Алгоритм был довольно условный, и сохранилась даже запись партии, где "бумажная машина" Тьюринга проиграла одному из его коллег.

Рис. 1.2. Алан Тьюринг (1912—1954)

Нужно отметить, что это была не единственная попытка изобрести алгоритм шахматной игры. Особенно примечательна история Ботвинника. Сам факт, что Ботвинник, будучи чемпионом мира и автором многочисленных работ по теории шахмат, воспитавшим целую плеяду чемпионов мира, верил, что существует алгоритм этой игры, заслуживает внимания. Но итог печален. Ботвинник как программист более 20 лет занимался созданием шахматной программы. Его программа так никогда и не заиграла. Кто-то даже сказал по этому поводу: "Ботвинник только думает, что он знает, как он думает". Алгоритм, возможно, существует, но он чрезвычайно сложен и, может быть, так никогда и не будет найден. Шахматы — не простая игра. В ней на любое правило всегда найдется исключение.

Примерно в это же время, в 1951 году, математик Клод Шеннон пишет свою первую статью о программировании шахмат. Он писал: "Хотя, возможно, это и не имеет никакого практического значения, сам вопрос является, теоретически, интересным, и будем надеяться, что решение этой проблемы послужит толчком для решения других проблем аналогичной природы и большого значения". Шеннон отмечает теоретическое существование лучшего хода в шахматах и практическую невозможность найти его. Он описывает две стратегии поиска лучшего хода, обе основываются на эвристической функции оценки конечных точек:

- тип А — перебор всех возможных ходов на фиксированную глубину, с вызовом в конце оценочной функции (т. к. невозможно осуществить перебор до конца);

- тип B — выполняет только выборочное расширение определенных строк, используя накопленные шахматные знания, чтобы подрезать неинтересные ветви. Например, программа Тьюринга только расширяла строки, включая взятия.

Все это выглядит несколько грубо и схематично с сегодняшних позиций, но принцип двух подходов (или их синтеза) сохраняется и в настоящее время. Шеннон писал, что тип B, несомненно, лучший, потому что больше похож на мышление человека. В чистом виде эти методы сегодня не используются, но, в зависимости от предпочтений автора конкретной программы, делается уклон в сторону одного или другого. Следующий шаг в развитии шахматных программ был сделан в ядерной лаборатории Лос-Аламоса в 1952 году, на компьютере MANIAC1 (11 кГц — тактовая частота, 600 слов память !!!). Для него была написана шахматная программа игры на доске 6×6, без участия слонов. Машина считала 4 полухода и тратила на это 12 минут. Известно, что этот компьютер сыграл одну партию против сильного шахматиста, она длилась 10 часов и закончилась победой шахматиста. Еще одна партия была сыграна против недавно научившейся играть в шахматы девушки. Машина победила на 23 ходу. Сейчас это выглядит смешно, но для своего времени это было целое достижение.

1957 год — на IBM704 (42 кГц, 7 Кбайт — память) был реализован тип B программы на полной доске, с участием всех фигур. Машина считала 4 полухода за 8 минут. Уровень игры — любительский.

1962 год — Ньюэл, Саймон и Шоу открывают алгоритм, получивший название alpha-beta. Он давал результат не хуже, чем полный перебор, не исследуя все варианты. В нем не требовалось никаких специальных шахматных знаний, и он мог быть применен для решения любой переборной задачи. Суть алгоритма, вкратце, в том, что в каждой строке игры, для белых и для черных отслеживались их максимальные результаты, и если в некоторой точке черные уже получили результат, сравнявшийся с максимумом белых, достигнутым до этого, то дальше перебирать нет смысла. Когда перебор вернется в точку, где был достигнут максимум белых, результат, все равно, будет отвергнут, т. к. у белых в этой точке есть уже не худший ход. Подробно эта техника будет описана далее. Наиболее известная работа по этой теме — статья Кнута и Мура в 1976 году. Считается, что этот алгоритм ранее открыл советский ученый Брудно. Хочется добавить, что основная заслуга этих людей в том, что они были первыми. Этот алгоритм великое множество раз открывался заново людьми, не подозревавшими о его существовании. Он очевиден. С его помощью можно считать значительно быстрее, но все еще недостаточно быстро. Тем не менее в основе всех современных шахматных программ лежит одна из усовершенствованных версий данного алгоритма. Примерно до 1973 года все шахматные программы были типа B. Они главным образом основывались на генераторах правдоподобных перемещений, отсекая статической оценкой малоправдоподобные. С появлени-

ем более мощных процессоров программисты стали переключаться на тип A. Первыми ласточками были Teach и Chess4. Это были программы "грубой силы". Как только они достигали глубины 5 полуходов средней стадии игры, они начинали побеждать в соревнованиях с программами типа B.

1975 год — Роберт Хьят начинает разрабатывать CrayBlitz, который долгое время был самой быстрой шахматной программой и в период с 1983 по 1989 гг. — мировым чемпионом среди шахматных программ. Он искал примерно 40—50 тыс. позиций в секунду (в 1983 году), что для своего времени было большим достижением.

1977 год — Томпсон и Кондон из лаборатории Bell создают первый специализированный шахматный компьютер. Дело в том, что скорость просчета зависит, помимо всего прочего, и от того, насколько программа задерживается в каждой позиции. Наиболее разорительными по времени являются генераторы ходов, функция оценки позиции, детекторы шахов и атаки и пр. Возникла простая идея, не дожидаясь увеличения мощности процессоров, реализовать эти части программы аппаратно. Иногда вещи, которые довольно сложно реализовать программно, очень просто решаются на уровне регистров процессора. Лучшие компьютеры того времени могли исследовать до 5 тысяч позиций в секунду, машина же Кена Томпсона, которую назвали Belle, обрабатывала 180 тысяч в секунду. Belle мог продумывать позицию на 8—9 полуходов вперед, что ставило его на уровень мастера. Он побеждал на многих компьютерных шахматных турнирах в период с 1980 по 1983 год. Несмотря на то, что специализированное железо на порядок быстрее, чем обычная машина, программа CrayBlitz на сверхсовременной тогда машине CrayXMPs (стоимостью 60 миллионов долларов) все равно играла лучше. В России подобные попытки были предприняты даже немного раньше: в конце 1960-х годов одна из отечественных машин М-20 была немного модернизирована, в нее были вставлены специализированные шахматные команды. В середине 1980-х годов Ганс Берлинер, компьютерный ученый университета Карнеги-Меллон, усовершенствовал компьютер Кена Томпсона. Был создан HiTech с параллельно действовавшими 64 шахматными чипами нового поколения. CrayBlitz, однако, все равно у него выиграл на всемирном чемпионате среди компьютеров в 1986 году. Вскоре после этого ученики Берлинера, студенты Хэй, Кемпбелл и др., разработали свой собственный специализированный шахматный компьютер Chip Test (он считал только материал), а позже и DeepThought (предшественник DeepBlue). DeepThought содержал 250 специализированных чипов, два процессора, был способен просчитать 750 тыс. позиций в секунду и вести перебор на 10 полуходов. Среди его достижений была победа над гроссмейстером Джудит Полгар. Однако Каспаров выиграл у экспериментальной шестипроцессорной версии DeepThought, которая искала более 2 млн позиций в секунду. Позже эта же группа, уже в расширенном составе, разработала DeepBlue. Таким образом, DeepBlue, скандально выигравший после модернизации второй матч у Каспарова (рис. 1.3), начинался как студенческая разработка.

Интересно было группе способных студентов пробовать свои силы, да и тема диплома была отличная. После создания DeepThought группу разработчиков заметил маркетинговый департамент фирмы IBM и обратился к ней с предложением. Результатом стали DeepBlue и DeepBlue2. Таким образом, DeepBlue — это результат более чем десятилетней работы очень способной группы студентов, при поддержке гроссмейстеров и миллионов IBM. Эта машина, помимо специализированного компьютерного железа, использовала распараллеливание alpha-beta перебора.

Рис. 1.3. Гарри Каспаров в решающем матче с DeepBlue

Оценочная функция DeepBlue не использует ни баз данных, известных из истории вариантов (кроме баз данных эндшпилей), ни генетических алгоритмов, ни нейронных сетей. Вместо этого, DeepBlue оценивает аппаратно приблизительно 6 тыс. специфических для шахмат показателей. Как производилась настройка этого множества показателей? В основном, вручную. Гроссмейстер Джон Бенджамен играл с машиной, пока не обнаруживал, что она неверно оценивает ту или иную позицию. После этого весь коллектив обсуждал проблему и изменял некоторые коэффициенты в оценочной функции. За большие деньги можно заниматься и не таким делом!

Немного об архитектуре DeepBlue. Основу машины представляет собой сервер RS6000SP. В сервере имеется 32 процессора, один объявлен главным. К каждому процессору подключено до 16 специализированных шахматных чипов — итого 512 на всю машину. Первые 4 полухода перебираются главным процессором. После этого он передает работу остальным 31, и они перебирают еще 4 полухода. Программа написана на обычном языке C, и использует алгоритм NegaScout. Предусмотрены выборочные продления некоторых вариантов, так что 4 полухода могут значительно растянуться. Некоторые критерии для продлений в DeepBlue:

- если значение оценочной функции для данного хода значительно выше, чем для остальных;

- по угрозе;
- по влиянию;
- проходные пешки;
- некоторые варианты, выходящие за границы alpha-beta (видимо, это так называемая сингулярная эвристика продлений, или ожидание второй отсечки за beta).

Система этих продлений описана несколько туманно. Такое впечатление, что продлевается все мыслимое и немыслимое. После основных процессоров работу перехватывают специализированные шахматные чипы. В них используются разработанные Кеном Томпсоном аппаратные средства для генерации перемещений, и осуществляется их сортировка. Аппаратно реализован детектор шахов (он работает один такт), контроль X-rays, вилок, перегрузок фигур и пр. Коэффициенты оценочной функции загружаются с диска, что дает возможность вести их настройку. Весь комплекс обрабатывает миллиард или более позиций в секунду.

Но вернемся к шахматным программам. Ричард Лэнг, пишущий исключительно на ассемблере (включая графическую часть!), сделал очень сильную программу выборочного поиска Genius. Любителям шахмат эта программа известна по шахматным компьютерам "Мефисто" с мощным специализированным процессором, в память которых зашит Genius. После проигрыша Каспарова в матче из двух партий (блиц) в 1995 году большинство ведущих гроссмейстеров потребовало запретить участие компьютеров в турнирах людей. Genius и сегодня участвует в мировых компьютерных первенствах и стабильно держит 5—6 место. Из особенностей Genius известно, что он просматривает шахи (особенно если у противника есть единственный ответ) на очень большую глубину.

Нужно отметить Hiarcs Марка Униаке, играющего в сильные позиционные шахматы. Hiarcs долгое время был мировым чемпионом и сегодня входит в тройку лучших программ.

Из отечественных программ нужно отметить "Каиссу" (Битман, Арлазоров, Адельсон-Вельский). Это была революционная для своего времени программа (1970-е годы). Она осуществляла с помощью итеративных углублений перебор на 7 полуходов плюс довольно сложные форсированные варианты. Использовались фильтры ходов. На обдумывание одного хода уходило примерно полтора часа. В матче с американцами программа выиграла буквально в самом начале партии, найдя очень глубоко форсированный мат. Американцы не могли поверить, что программа могла считать так глубоко, но им показали, что вся комбинация состоит из форсированных ходов: взятий и шахов. К сожалению, развития эта победа не получила, и самые сильные программы стали появляться за рубежом. Исключение составляет "Мираж" (1992, 1998, Рыбинкин-Шпеер). Он по сей день остается чемпионом России. Чемпионаты, правда, уже не проводятся. Это очень интересная

программа выборочного поиска (подробностей я, к сожалению, не знаю). Известно, что "Мираж" считает не все, но достаточно глубоко. Он участвовал в мировых первенствах, но первого места (как и последнего) не занимал. Играет он очень интересно, но стабильно сильную игру не показывает. Известны его разгромные победы над Fritz и Hiarcs. В настоящее время Сергей Марков, автор SmarThink, утверждает, что его программа играет сильнее, чем "Мираж". Я лично нисколько не сомневаюсь, что некоторые варианты она просчитывает лучше, но надо еще доказать, что она стабильно сильнее играет. "Мираж" все-таки очень удачная программа.

Из развития шахматных алгоритмов хочется отметить появление эвристики пустого хода (NullMove). Идея состоит в том, что в большинстве случаев, если мы пропускаем ход (передаем очередь хода противнику и перебираем на меньшую глубину, на два или более полухода), то теряем, а не приобретаем. Если после такого пропуска наша оценка все еще больше beta, то перебор можно прервать — это неинтересная часть дерева игры. Точное время появления этой эвристики неизвестно. В 1986 году на мировом первенстве в Германии Дон Бил участвовал с программой, которая использовала технику недействительного перемещения в его статическом поиске. В течение турнира Франц Морш заинтересовался этой идеей и спустя несколько лет осуществил ее в программе Fritz. Ускорение получалось очень значительное, а результат очень точным. С середины 1990-х годов недействительное перемещение стало использоваться практически во всех шахматных программах, (правда, подход к его применению очень различный).

Следующее усовершенствование было сделано Эрнст Хайнц в его программе DarkThough и называлось Futility pruning и Razoring. Это выборочное отсечение определенных пограничных ветвей дерева перебора, без их предварительного исследования. Отсекаемая позиция должна удовлетворять определенным условиям. Данные методы не вносят практически никакой погрешности, но есть люди, не признающие их. Они никак не могут понять, что считать надо очень глубоко, а рост дерева выражается экспоненциальной зависимостью и "засадит" любой мыслимый и немыслимый процессор.

Некоторые приемы программирования

Поклонники языков C и Паскаль могут пропустить этот раздел. В нем рассказывается о некоторых тривиальных приемах программирования, которые, тем не менее, полезно знать или освежить в памяти перед чтением книги. Это, во-первых, рекурсия.

Рекурсия

Кто получил специальное образование (повезло с преподавателем) или сам занимался этим вопросом, тот, несомненно, знает преимущества рекурсив-

ных алгоритмов. В обычных учебниках им отводится до смешного мало места или не отводится вообще. Популярно следующее замечание: "Любая задача, решенная с помощью рекурсии, может быть решена и обычным способом". Это так, если моделировать стек программы. Некоторые задачи, тем не менее, практически невозможно решить без рекурсии. Объем кода неимоверно возрастает, и без всякой пользы.

Рекурсия — это вызов из функции самой себя. Это допустимо даже в Бейсике. Функция вызывает сама себя, при этом все переменные (стековые), кроме статических, задаются заново. Если у нас есть некоторая функция `Foo` и переменная `dummy`, то `dummy` всегда новая при следующем вызове функции.

Приведем пример "пустой" рекурсии. Функция вызывает сама себя и не совершает никакой полезной работы. Единственное, нужно ограничить количество рекурсивных вызовов некоторым реальным числом:

```
void Foo(int callNumber)
{
if(callNumber <= 0) return;
Foo(callNumber-1);
}
```

Пример первого вызова функции:

```
Foo(10);
```

Функция `Foo` вызывает себя 10 раз, и программа завершается. Значение `callNumber` уменьшается при каждом вызове, и если равно нулю, дальнейших рекурсивных вызовов нет. Переменная `callNumber` обозначает глубину рекурсии и обычно называется `depth`. Ключевое слово **`void`** обозначает, что функция не возвращает никакого значения. В языке C нет специально зарезервированного слова `function`. Оно подразумевается. В C++ мы можем не писать **`void`** (его можно опустить и в некоторых версиях языка C). Тогда будет подразумеваться, что функция возвращает целочисленный тип со знаком — **`int`**. Размерность типа **`int`** равна разрядности регистра аккумулятора (`EAX` или `AX`) процессора. Для 16-разрядных программ это 16, для 32-разрядных — 32 бита. Напишем:

```
Foo(int depth)
  {
if(depth > 0) Foo(depth-1);
return 0;
  }
```

Можно, без привычки, не сразу и догадаться, что перед нами сигнатура функции. Возвращаемое значение 0, в данном случае, неинформативно. Некоторые компиляторы позволяют опускать его. Это ненужные тонкости. Желательно, если функция не возвращает значения, писать явно **`void`**.

Это же относится и к списку аргументов функции. Если аргументов нет, то в C нужно обязательно указать:

```
void Foo(void);
```

В C++ слово `void` можно опустить и просто написать:

```
void Foo();
```

Давайте заставим нашу рекурсивную функцию совершить какую-нибудь полезную работу. Например, вычислить возведение числа в степень.

```
int Foo(int value, int n)
  {
if(n == 0) return 1;
return value * Foo(value,n-1);
  }
```

Описанная функция не вычисляет отрицательные степени. Это как раз неудачный пример рекурсии, и задача может быть легко решена без нее:

```
int Foo(int value, int n)
  {
int tmp = 1;
for(;n > 0; n--)
tmp = tmp * value;
return tmp;
  }
```

В некоторых случаях рекурсия, тем не менее, оказывается очень полезной. Вот пример.

Ханойские башни

Имеется 3 стержня: A, B и C. На стержень A нанизано n дисков так, что диск с меньшим диаметром всегда лежит поверх диска с большим. Говоря другими словами, на стержне A пирамида из дисков, сужающаяся вверх. Требуется переместить все диски на стержень B. Стержень C можно использовать в качестве вспомогательного. Основное условие состоит в том, что диск с большим диаметром не должен находиться поверх диска с меньшим.

Как это сделать? Предположим, что мы умеем перекладывать пирамиду из n−1 дисков. Как, мы не знаем, но умеем. Тогда нам нужно переместить первые n−1 дисков с A на C. Затем перенести самое большое кольцо с A на B. Оно окажется внизу. Затем n−1 дисков нужно переместить с C на B. Готово. Мы только задали закон развития алгоритма. Как он будет выполнять-

ся в действительности — очень любопытно посмотреть по распечатке результата. Если дисков много, задача получается очень головоломной.

```
#include <stdio.h>
void Ring(int n, char a, char b, char c)
{
if(n <= 0) return;
Ring(n-1,a,c,b);
printf("диск %d   %c -> %c \n",   n,a,b);
Ring(n-1,c,b,a);
}

//основная программа
int main(void)
{
Ring(3,'a','b','c');
return 0; //код завершения программы
}
```

Мы вызываем функцию `Ring`, чтобы переложить только 3 кольца. Если увеличить это число, то процедура перекладывания будет очень замысловатой.

Директивой `#include` в программу включен заголовочный файл `<stdio.h>`. Он содержит стандартные библиотеки ввода/вывода языка C, и описание функции `printf` находится именно в нем. Текст программы поступит сначала на вход препроцессора языка C. Он просмотрит данный файл и вставит программный код файла stdio.h точно в то место, где он объявлен директивой `#include`. Препроцессору все равно, какой файл вставить в это место. Было бы указано его название, и был бы он на диске в доступном каталоге.

Когда код программы поступит на вход компилятора, он будет значительно больше приведенного, но это нас не волнует, в данном случае. Компилятор при трансляции кода делает один проход, поэтому он должен встретить предварительное описание функции `printf` перед тем, как вызов данной функции встретится в программе. Иначе компилятор не будет знать, что это за функция, и каков у нее список аргументов. Наша небольшая программа содержится в файле с расширением c. При трансляции кода компилятором получится объектный файл с расширением obj. Он должен быть связан компоновщиком (например, `link`) с другими объектными файлами, если они есть в проекте. В результате мы получим выполняемый файл с расширением exe, который и является исполняемой программой. В нашем случае компоновщик должен будет найти библиотечный объектный файл (name.obj или name.lib) и взять оттуда сам код функции `printf`. В файле stdio.h содержится только ее предварительное (forward) описание. Мы можем и не включать файл stdio.h, а написать в начале программы:

```
extern printf(...);
```

Сигнатуру функции `printf` можно посмотреть в вашем файле stdio.h. Это обычный текстовый файл. Сигнатура функции `printf` непростая, и лучше оставить все это на совести разработчиков компилятора.

В проекте может быть несколько файлов с расширением c. Компилятор для каждого создаст свой объектный файл. Единственное, что нужно помнить, что это функция `main` является точкой начала исполнения программы, и она должна быть одна в каком-либо файле. В каком — безразлично.

Зачем нужно много файлов? Это очень удобно. Некоторые файлы могут содержать пользовательские библиотечные функции, и их можно включать в разные проекты. Как вызвать из одного файла функцию, содержащуюся в другом? Для этого нужно воспользоваться директивой **`extern`**. Если мы хотим ограничить область видимости некоторой функции одним файлом (или переменной), мы должны перед ее сигнатурой написать ключевое слово **`static`**. В нашем примере функция `Ring` и основная функция `main` были в одном файле. Давайте поместим функцию `Ring` в отдельный файл. Тогда основной файл программы будет содержать только функцию `main`. Перед использованием функции `Ring` нужно написать:

```
extern Ring(int n,char a, char b, char c);
```

Когда компилятор встретит это описание, он ничего не будет знать об устройстве функции `Ring`, потому что каждый файл компилируется отдельно. Он будет только знать сигнатуру данной функции, чтобы осуществить ее вызов. При этом компилятору не важны названия аргументов, важен их тип и порядок. Мы могли бы написать:

```
extern Rint(int,char,char,char);
```

Компоновщик после компиляции установит необходимые связи. Если он не найдет функцию `Ring` ни в одном файле проекта, он сообщит об ошибке компоновки. Директива **`extern`** может использоваться и для объявления внешних переменных. Мы можем написать в одном файле:

```
extern int dummy;
```

А сама переменная `dummy` будет находиться в другом файле.

```
int dummy = 0;
```

Каждый раз использовать директиву **`extern`** не всегда удобно. У нас может быть большой проект и много файлов. Мы можем собрать все внешние объявления (**`extern`**) в одном или нескольких файлах с расширением h и включить их директивой `#include`. Давайте создадим для нашего случая простой файл (*.h) и включим туда описание внешней функции `Ring`.

```
//файл MyHeader.h
#idndef MY_HEADER
```

```
#define MY_HEADER
extern Ring(int,char,char,char);
#endif
```

Файл main.c тогда будет выглядеть так:

```
#include "MyHeader.h"
int main(void)
{
Ring(3,'a','b','c');
return 0;
}
```

Сама функция `Ring` будет находиться в третьем файле с расширением c, название которого совершенно не важно. Обратите внимание, что в файле MyHeader.h содержится только описание функции `Ring`. В каком она файле — не указано. В одном из файлов проекта. Директиву `extern` можно опустить для описания внешней функции в файле (*.h). Это подразумевается. В языке C значком # начинаются директивы препроцессора. Это не компилятор, это отдельная программа, которая совершает предварительную обработку текста программы. Компилятор не встретит в программе уже ни одной директивы, начинающейся значком #. В нашем файле MyHeader.h есть несколько директив препроцессора. Их можно опустить, и тогда файл будет выглядеть:

```
//////////////////////////////////
void Foo(int,char,char,char);
//////////////////////////////////
```

Зачем они нужны? Дело в том, что данный файл может быть включен в текст программы несколько раз. Как это может быть? Допустим, у нас несколько заголовочных файлов (*.h). В один файл может быть включен другой заголовочный файл, а потом оба, по ошибке, включены в третий заголовочный файл. Это происходит довольно часто. Как этого избежать? Для этого мы используем директиву препроцессора, проверяющую условие. Она аналогична оператору `if` языка программирования, только пишется `#if`, например:

- директива `#ifdef` — если определена;
- директива `#ifndef` — если не определена;
- директива `#define` вводит константу (символическое имя препроцессора).

В нашем случае мы написали:

```
#define MY_HEADER
```

`MY_HEADER` — это вымышленное символическое имя препроцессора. Если файл MyHeader.h был включен препроцессором хоть один раз при трансляции кода для конкретного файла, то это имя препроцессор запомнил. Если встретится повторное включение файла MyHeader.h, то препроцессор проверит условие:

```
#ifndef MY_HEADER
```

Так как оно ложно, весь текст до директивы `#endif` будет пропущен, и повторного включения файла MyHeader.h не произойдет. Мы должны понимать, что препроцессору все равно, что включать и сколько раз. Если мы напишем 10 раз `#include "MyHeader.h"` и не примем мер, то препроцессор включит этот файл 10 раз в место, указанное директивой `#include`. Ошибку выдаст компилятор. Он 10 раз встретит предварительное описание функции `Ring` и откажется дальше компилировать. Если разобраться, в этом нет ничего сложного. Это обычные макросы. Нам может быть неудобно много раз выписывать один и тот же код, и мы можем его оформить в виде макроса. Потом в тексте программы, вместо того чтобы нудно выписывать одно и то же, мы можем просто написать имя макроса, и препроцессор расширит его нужным кодом. Например, у нас есть массив из 10-ти чисел, и нам нужно его много раз обнулять или присваивать какие-либо значения. Мы можем написать один раз макрос, а потом использовать в программе имя макроса, например:

```
//инициализация массива data[10]
#define INIT_ARRAY \
data[0] = 1; data[1] = 2;\
data[3] = 4;.... data[9] = 512;
```

Содержимое файла (*.h) вставляется точно так же препроцессором, и мы должны написать в этом файле то, что хотим видеть в этом месте программы. Как некоторый рекурсивный казус, мы можем включить в файл MyHeader.h в самом конце сам этот файл `#include "MyHeader.h"`. Что произойдет? Конечно, препроцессор должен заметить ошибку, иначе он будет расширять файл MeHeader.h самим собой, пока хватит памяти, или впадет в бесконечный цикл.

Задача о ферзях

Найти способ расстановки 8 ферзей на доске 8×8 так, чтобы они не били друг друга. Данная задача решается методом перебора с помощью рекурсии. Прежде чем поставить на доску очередного ферзя, нужно проверить, не бьет ли его какой другой ферзь. Если мы будем каждый раз сканировать все возможные перемещения всех ферзей, то это достаточно неэффективно. Можно ускорить проверку, если обратить внимание на один факт. Ферзь, как

известно, ходит во всех направлениях, по всем диагоналям, вертикалям и горизонталям. С вертикалями и горизонталями все понятно: если был хоть один ферзь с координатой X или Y такой, как у нашего ферзя, то данное поле бьется. Как быть с диагоналями? Если диагональ восходящая (/), то сумма X и Y для всех клеток диагонали одна. Если нисходящая (\), то разность X — Y для всех клеток одинакова. Мы можем иметь некоторые множества, обнуленные в начале вычисления:

1. Множество X.
2. Y.
3. X+Y.
4. X−Y.

Чтобы узнать, бьется ли данное поле каким-либо ферзем, мы должны проверить клетки массивов (множеств) 1, 2, 3, 4. Для массива 1 это будет клетка с индексом X, для массива 2 — Y, для 3 — (X+Y), для 4 — (X−Y+8). Все эти клетки должны быть нулевыми. Если хоть одна клетка 1, то поле бьется. Если мы поставили ферзя в какую-то клетку, то в соответствующие ячейки множеств должны записать 1. Если отменили ход — должны восстановить старое значение ячеек. Массив 4 имеет добавочный индекс 8 потому, что X−Y не может быть меньше 0, а в языке C индексация массивов от 0. Если мы поставили последнего ферзя, то конец перебора и вывод результата. Так как мы имеем дело с частным вырожденным случаем, то вместо 4 множеств можно использовать 3, а координату X увеличивать пошагово. Это гарантирует нам, что не будет двух ферзей с одинаковой координатой X. Старые значения множеств можно также не сохранять и не восстанавливать, т. к. если мы поставили ферзя на клетку, то это гарантия того, что в соответствующих ячейках массивов были нули. Вы можете попробовать написать эту программу сами. Это довольно занятно. Можно попытаться получить все возможные размещения ферзей.

Я же хочу обратить внимание на некоторый аспект этой задачи применительно к программированию шахмат. Допустим, нам нужно определить, бьется ли поле (X,Y) фигурами противника. Как это сделать? Существует множество способов, но применительно к данному примеру мы рассмотрим один. Допустим, у нас есть 4 массива со значениями в ячейках, соответствующих дальнобойным фигурам противника. В реальной программе это может делаться пошагово. Поставили фигуру (сделан ход) — увеличили соответствующие ячейки на единицу, убрали — уменьшили. Если в первом массиве у нас не 0 в ячейке с индексом X, возможно, в этой вертикали есть ладья или ферзь. Мы должны просканировать два луча в обоих направлениях до первой фигуры. Если в третьем массиве в ячейке с индексом (X+Y) не 0, то, возможно, на восходящей (/) диагонали нас бьет ферзь или слон противника. Нужно тоже просканировать два луча. Приращение при построении линии известно сразу. Для пешек нужно просто проверить две

соседние клетки по диагонали. Несколько сложнее с королем и конем. Проверять соседние 8 клеток, в надежде найти там короля противника, и еще 8, в надежде найти коня, как-то не хочется. Но об этом несколько позднее.

Локальные функции

Локальной или вложенной называется процедура или функция, описанная внутри другой процедуры (функции). Она может иметь свои локальные переменные, а может и обращаться к переменным функции, внутри которой она описана. Вызывать ее можно только из функции, внутри которой она описана, т. е. область видимости ограничена родительской функцией.

Основным недостатком языка C при программировании шахмат является именно отсутствие в нем локальных функций. Вот пример локальной функции (процедуры) на паскале:

```
procedure Foo(arg1,arg2:integer);
procedure SubFoo(arg3,arg4:integer);
begin
         {...}
end;
begin
     {...}
end;
```

В функции `Foo` можно вызвать процедуру `SubFoo`, которая может обращаться к переменным `arg1`, `arg2` вызывающей ее функции `Foo`. Почему в C нет локальных функций? Трудно сказать. В языке C первоначально много чего не было. Видимо, основная причина заключается в том, что это считается несистемной вещью, т. к., чтобы обратиться к переменным родительской функции, `SubFoo` должна извлечь из стека регистр `BP(EBP)`, а проще говоря, указатель на эти переменные. Тем не менее это очень удобный прием. В некоторых версиях компиляторов C можно эмулировать локальную функцию на ассемблере (если она не имеет своих аргументов):

```
void Foo(int arg1, int arg2)
  {
SubFoo: //метка
         {...}
_asm ret; //возврат из подпрограммы
{...} //инструкции функции Foo
_asm call SubFoo; //вызов вложенной подпрограммы
}//function
```

Этот прием, тем не менее, является довольно опасным. Нужно очень четко представлять, какие регистры нужно сохранить в локальной процедуре, и, кроме того, разработчики некоторых компиляторов очень активизируются в данных случаях, и нет гарантии, что это будет работать, раз не предусмотрено синтаксисом языка. Гораздо проще оформить переменные (стековые) функции `Foo` в виде структуры, а в `SubFoo` передавать указатель на эту структуру.

```
typedef struct{
int arg1,arg2;
}Targ.*Parg;
void SubFoo(Parg arg)
   {
     {...}
   }
vois Foo(Parg arg)
   {
     {...}
SubFoo(arg);
   }
```

В некоторых случаях такой прием бывает чрезвычайно удобен. Функция `Foo` может быть очень большой и содержать множественные вызовы `SubFoo`, которая, в свою очередь, вызывает вложенные функции следующего уровня. Передавать каждый раз огромный список параметров очень накладно, и, кроме того, локальные функции должны иметь возможность изменять эти переменные. К сожалению, данный прием не улучшает читабельность программы. Когда-нибудь в C появятся локальные функции (как и нормальный строковый тип данных).

Игра "Уголки"

Для иллюстрации всего вышесказанного приведем небольшой пример. На игровом поле 8×8 находятся 12 черных шашек (в левом верхнем углу) и 12 белых (в правом нижнем). Нужно занять исходные позиции противника. Черные должны стремиться занять правый нижний угол, а белые — левый верхний. Шашки ходят во всех направлениях на одну клетку (кроме диагоналей). Шашка может прыгать через другие шашки в любых направлениях (кроме диагоналей).

Для решения этой задачи используется перебор. Ход противника не учитывается. Если говорить про моделирование поведения противника, то мы моделируем "болвана". Тем не менее, т. к. взятий нет, это неплохо работает на практике. Используется некоторый массив, где в каждой клетке нарисовано, куда фигуре лучше пойти. Просчет мы ведем только для черных. Таким об-

разом, числа массива минимальные в верхнем левом углу и максимальные в правом нижнем. Теоретически, массив оценок для черных должен быть следующий:

```
1, 2, 3, 4, 5, 6, 7, 8,
9,10,11,12,13,14,15,16,
17,18,19,20,21,22,23,24,
25,26,27,28,29,30,31,32,
33,34,35,36,37,38,39,40,
41,42,43,44,45,46,47,48,
49,50,51,52,53,54,55,56,
57,58,59,60,61,62,63,64
```

Я использовал другие числа, чтобы программа более явно стремилась в правый нижний угол. Дело в том, что мы не учитываем ход противника (для него массив инвертирован), и у нас нет времени на вычисления. В "Уголках" программа должна ходить практически сразу. Если она будет думать по 30 секунд над тривиальной задачей и вычислять позиционные преимущества, это будет выглядеть смешно. Мы считаем на 3 хода (только своих). Если бы мы учитывали ход противника, то задача ничуть не уступала бы шахматам. Потребовалась бы большая глубина перебора (чтобы было хотя бы 3 своих полухода, нужно суммарно иметь 6: 3 своих и 3 противника). Пришлось бы прибегать к сложным программным решениям, а для такой простой программы это выглядело бы нелепо. На практике играла бы программа не лучше (может, и хуже). Данный прием пропуска хода используется и в шахматах, но несколько по-другому. Суть в том, что если мы делаем подряд два хода, это засчитывается как наш возможный максимум. Если противник сделает ход, то результат может быть только меньше для нас. Здесь мы считаем на 3 хода и, следовательно, пропускаем 2 хода противника (полухода). Получается, что мы играем не с "болваном", а с "суперболваном". Программа ходит так, как если бы противник 2 раза не походил. Но т. к. игра построена в основном на "зевках", это работает.

Есть еще один интересный момент в программе. Это определение "прыгающих" ходов шашки. Шашка начинает движение из своей стартовой точки и может выписывать довольно замысловатые траектории. При нахождении возможного пути мы пользуемся следующим правилом: шашка может сделать любой легальный ход, за исключением тех клеток, на которых она уже побывала. Это только при взятии перемещений шашки в одном узле. Для запоминания полей, на которых шашка уже побывала, мы используем множество, в котором запоминаем индекс клетки массива, где побывала шашка. В языке C типа данных "множество" нет, и приходится конструировать его самим. Это делается чрезвычайно просто. Решение довольно любопытное и может применяться в других программах. Наше множество представлено массивом 16 байт. В каждом байте — 8 бит, следовательно, в мно-

жестве может быть запомнено 16 × 8 = 128 элементов (значений от 0 до 127). Если элемент (например, 10) есть, в множестве соответствующий бит установлен в 1. Если нет — в 0. Для определения бита, соответствующего данному числу, мы должны найти номер байта и номер бита в этом байте. Это делается очень просто:

```
byteNumber = N / 8;   //результат деления
bitNumber  = N % 8;   //остаток от деления
```

Игровое поле в уголках 8 × 8. Мы используем несколько больший массив: (10,10). Края массива (лишние поля) заполнены флагом выхода за пределы поля. Таким образом, не нужны проверки выхода за пределы. Мы всегда наращиваем координату на одну клетку, и если в ней флаг выхода за пределы игрового поля, то это значит, что мы вышли за пределы поля 8 × 8.

Листинг 1.1. Игра "Уголки"

```
CopyRight ©  2004 by Kornilov Evgeny e_k@sbor.net
//////////  DEFINITION ////////////////
#define B    1  //черная шашка
#define W    2  //белая
#define F    3  //флаг выхода за пределы доски
//стартовая позиция
int pos[10*10] =
{
F,F,F,F,F,F,F,F,F,F,
F,B,B,B,B,0,0,0,0,F,
F,B,B,B,B,0,0,0,0,F,
F,B,B,B,B,0,0,0,0,F,
F,0,0,0,0,0,0,0,0,F,
F,0,0,0,0,0,0,0,0,F,
F,0,0,0,0,W,W,W,W,F,
F,0,0,0,0,W,W,W,W,F,
F,0,0,0,0,W,W,W,W,F,
F,F,F,F,F,F,F,F,F,F
};
//куда лучше ходить
int StTab[10*10] =
{
0, 0,0,0,0,0,0,0,0,0,
0, 1,2,3,4,5,6,7,8,0,
0, 9,10,11,12,13,14,15,16,0,
0, 10,17,18,19,20,21,22,23,0,
0, 11,18,24,25,26,27,28,29,0,
```

```
0, 12,19,25,30,31,32,33,34,0,
0, 13,20,26,31,401,402,403,404,0,
0, 14,21,27,32,402,405,406,407,0,
0, 15,22,28,33,403,406,408,409,0,
0, 0,0,0,0,0,0,0,0,0
};
//стартовый список фигур машины (черных)
int PieceTab[12] = {11,12,13,14,
21,22,23,24,
31,32,33,34};
//номера конечных позиций
int EndTab[12] = {65,66,67,68,
75,76,77,78,
85,86,87,88};
//приращения при движении шашки
int DelTab[4] = {-1,1,-10,10};
//описатель перемещения
typedef struct{
int source,dest;
}TMove,*PMove;
////////////////тип – множество//////////
typedef union{
unsigned char data[16];
long __data[4];
}TSet,*PSet;
//обнуляет множество
#define Clear(set)\
(set).data[0]=0; (set).data[1]=0; (set).data[2] = 0; (set).data[3]=0;
//включает элемент n в множество set
#define Include(set,n)\
(set).data[(n) >> 3]   |=   (1 << ((n) & 7));
//возвращает не 0, если элемент n в множестве set
#define In(set,n)\
( (set).data[ (n) >> 3 ]    &   (1 << ((n) & 7)) )
////////////////////////////////////////////
//переменные при просчете
typedef struct{
int score,ply,max;
TMove bestMove;
TSet set;
}Targ,*Parg;
//максимальное значение при просчете
#define INFINITY 30000
```

```
//максимальная глубина просчета
#define MAX_PLY 3
//проверка на конец
int CheckEnd(void)
{
int n;
for(n = 0; n < 12; n++)
if(pos[ EndTab[n] ]  !=  B) return 0;
return 1;
}//function
//делает перемещение
#define MakeMove\
pos[move.source] = 0;\
pos[move.dest] = B;\
PieceTab[index] = move.dest;
//отменяет ход
#define UnMakeMove\
pos[move.dest] = 0;\
pos[move.source] = B;\
PieceTab[index] = move.source;
///////  FORWARD //////////
void SimpleMove(Parg,int);
void Move(Parg,int);
void CallSearch(Parg,PMove);
void Search(Parg);
//простой ход на клетку
void SimpleMove(Parg arg,int index)
{
int n;
TMove move;
move.source = PieceTab[index];
for(n = 0; n < 4; n++)
   {
move.dest = move.source + DelTab[n];
if(pos[move.dest] == 0)
       {
MakeMove
CallSearch(arg,&move);
UnMakeMove
       }
   }
}//function
//прыгает через шашку
void Move(Parg arg, int index)
```

```
{
int n,tmp;
TMove move;
for(n = 0; n < 4; n++)
   {
move.source = PieceTab[index];
move.dest = move.source + DelTab[n];
tmp = pos[move.dest];
if(tmp == 0  ||  tmp == F) continue;
move.dest += DelTab[n];
tmp = pos[move.dest];
if(tmp != 0) continue;
if(In(arg->set, move.dest)) continue;
//нашли — куда походить
Include(arg->set, move.dest);
MakeMove
CallSearch(arg,&move);
Move(arg,index);
UnMakeMove
}//for
}//function
//вызов просчета
void CallSearch(Parg arg, PMove move)
{
Targ nextArg;
nextArg.ply = arg->ply+1;
nextArg.score = arg->score  +
(StTab[move->dest] — StTab[move->source]);
Search(&nextArg);
//если нашли лучший ход
if(nextArg.max > arg->max)
    {
arg->max = nextArg.max;
arg->bestMove.source = move->source;
arg->bestMove.dest   = move->dest;
    }
}//function
//основная функция просчета
void Search(Parg arg)
{
int n;
if(arg->ply >= MAX_PLY)
     {
arg->max = arg->score;
```

```
return;
     }
if(CheckEnd())
     {
arg->max = INFINITY — arg->ply;
return;
     }
arg->max = -INFINITY;
//все перемещения на одну клетку
for(n = 0; n < 12; n++)
SimpleMove(arg,n);
//все перемещения через шашки
for(n = 0; n < 12; n++)
     {
//очистим множество
Clear(arg->set);
//занесем стартовую координату
Include(arg->set,PieceTab[n]);
Move(arg, n);
     }
}//function
//  пример первого вызова
/********
Targ arg;
int n;
memset(&arg,0,sizeof(Targ));
Search(&arg);
//изменение в позиции и в списке фигур
for(n = 0; n < 12; n++)
if(PieceTab[n] == arg.bestMove.source)
    {
PieceTab[n] = arg.bestMove.dest;
pos[arg.bestMove.source] = 0;
pos[arg.bestMove.dest] = B;
break;
    }
*********/
```

Грубое усилие и избирательность

Когда в 70-е годы прошлого века состоялся второй матч между русской "Каиссой" и американской Chess, то создатели американской программы сделали эпохальное заявление: "Мы собираемся заниматься выборочным

поиском, потому что полный перебор — это тупик". Такое или подобное заявление делали многие шахматные программисты на определенном этапе своей деятельности. И только очень немногие сделали приличную программу выборочного поиска. Среди них Ричард Лэнг с его программой Genius. В чем же трудность? Почему так мало удачных программ выборочного поиска? На этот вопрос можно ответить, но меня смущают некоторые вещи. Я недавно читал книгу по 3D-движкам и понял содержание только первых нескольких глав, да и то потому, что сам в свое время построил простейший 3D-движок. Остальная часть книги совершенно непонятна. Темы не раскрыты, а только затронуты, тон менторский, автор — явный зазнайка. В примерах программ к книге у него вращался (очень медленно) трехмерный зеленый бублик, и это приводилось в качестве достижения трехмерного моделирования. Если бы все читали только эту книгу, то не было бы ни Doom, ни Ducke, ни других хороших игрушек. Для кого была написана та книга? Видимо, для тех, кто и так все знает, а только хочет нечто освежить в памяти. Я все боюсь уподобиться этому автору. Для меня в alpha-beta алгоритме и его свойствах нет ничего сложного, потому что я долго занимался этим вопросом. Как дела обстоят у читателя? Да простят меня профессионалы, я буду много времени уделять тривиальным вещам и много раз повторяться. Я надеюсь, что при каждом повторе высветится та или иная грань алгоритма.

Почему только alpha-beta? Дело в том, что никакой альтернативы этому алгоритму не найдено, и это как искусственный интеллект в миниатюре, если мы считаем до конца или до такой большой глубины, что погрешностью можно пренебречь. Что значит считать до конца? В любой игре есть некоторый набор правил и есть условие, когда игра закончена. Вот и нужно считать до этого условия. Оно может никогда и не встретиться из-за повторов позиций. Что такое повторы? Это когда позиция повторяется в строке игры через несколько ходов. В шахматах, если позиция повторилась три раза в строке (проигранной), то оппонент может потребовать ничью. Например, есть некоторая строка:

```
a1b1 a8b8 b1a1 b8a8
```

Здесь белая ладья пошла с a1 на b1, а черная — с a8 на b8. Это 2 полухода. В следующих двух полуходах видно, что белая и черная ладья вернулись на свои места. Получился повтор позиции через 4 полухода. Если такая ситуация повторится 3 раза, то через 12 полуходов может быть предъявлена оценка `draw`. Оценка `draw` при просчете в большинстве случаев принимается равной нулю (ничья). Некоторые программисты делают оценку `draw` равной небольшой отрицательной величине, чтобы программа при просчете не зацикливалась из-за небольшого минуса в стратегической оценке. Что такое стратегическая оценка?

Оценка в шахматах может быть стратегической и материальной. Любой человек, учащийся играть в шахматы, должен представлять себе ценности фи-