sun® microsystems

Games, Demos *and* Other Pursuits: Beginner's Guide

Part No: 800-1270-03
Revision A of 17 February 1986

Credits and Trademarks

Contents

Figures

Preface

This document describes concisely the standard set of games and demos that run on Sun workstations.

Companion documents

Commands Reference Manual
SunCore Reference Manual
Pixrect Reference Manual
Setting Up Your UNIX Environment: Beginner's Guide
Self Help With Problems: Beginner's Guide
Windows and Window-Based Tools: Beginner's Guide
Doing More with UNIX: Beginner's Guide

1

Introduction

1 Introduction

Computer games and demos have been a tradition for as long as there have been computers. At Sun, we continue the tradition, offering the traditional UNIX† games and demos, and some new ones that we have developed.

Games are for amusement, or to learn about subjects not necessarily related to computers. *Demos* are attractive graphics programs that demonstrate the capacities of the system or help programmers learn, by example, to design and implement graphics displays of their own.

Do not assume that an executable version of any particular game or demo is available on your system. Sometimes, Sun distributes games and demos on a separate tape, or distributes only the source and data files for games and demos. If, after reading this manual, you have trouble finding or running a game or demo, contact your system administrator.

1.1. Games

Sun games come in three flavors: *dialogue games*, *terminal games*, and *graphics games*.

Running Games

To run a game, type the name of the game (and any arguments or options that are appropriate).

If that does not work, make sure your search path includes the `/usr/games` directory, or that you specify the pathname of the game explicitly. For example, type:

```
tutorial% adventure
```

to run the game `adventure` when you have `/usr/games` in the `set path=` entry of your `~/.login` or `~/.cshrc` file. Or, type:

```
tutorial% /usr/games/adventure
```

if you don't wish to put `/usr/games` in your search path.

† UNIX is a trademark of AT&T Bell Laboratories.

Note: If you *do* quit a game by typing CTRL-C, remember to reset the terminal afterwards, else your terminal may be confused.

You can exit from most games by typing CTRL-C; individual writeups for games list the exceptions.

1.2. Demos

Sun demos are either *Graphics Standards* or *Pixrect Graphics* demos. Demos run on workstations that have adequate graphics capabilities; see the descriptions of individual demos for details. Most demos run on all graphics devices, including Sun-1, Sun-2, and Sun-3 monochrome or color frame buffers (the entire screen), and black-and-white or color subwindows (partial screen).

To run a demo, type the name of the demo (and any arguments or options that are appropriate).

If that does not work, make sure your search path includes the `/usr/demo` directory, or that you specify the pathname of the game or demo explicitly. For example, type:

```
suncube
```

to run the demo `suncube` when you have `/usr/demo` in the `set path=` entry of your `.login` or `.cshrc` file. Or, type:

```
/usr/demo/suncube
```

if you don't wish to put `/usr/demo` in your search path.

You can exit from most demos by typing CTRL-C; individual writeups for demos list the exceptions.

Most demos allow you to direct the demo to a device other than that of your current environment. To change the display device, type the demo name followed by the option `-d /dev/`*devicename*. For example, to run the `suncube` demo on a color monitor, starting from a black-and-white monitor, type:

```
suncube -d /dev/cg0
```

When you start a demo from a black-and-white subwindow, to run on a color subwindow, the demo can't use the mouse. So, demos requiring the mouse, like `draw`, don't work when you start them in a black-and-white subwindow and display them in a color subwindow.

2

Question and Answer Games

2

Question and Answer Games

Question-and-answer games, or dialogue games, are the only games that run on teletypes. They also run on video terminals and graphics devices.

2.1. Adventure

`/usr/games/adventure`

The object of `adventure` is to locate and explore Colossal Cave, find the treasures hidden there, and bring them back to the building with you. The program describes itself to a point, but part of the game is to discover its rules.

type `quit` to terminate a game; type `suspend.` to save a game for later resumption. When you suspend a game, `adventure` creates an executable file with a filename that it prompts you for. `adventure` creates the saved game file in your current directory unless you specify otherwise. You must wait at least 45 minutes before playing the saved version; to execute it, just type the filename you specified.

Caveats

Saving a game creates a *large* executable file instead of just the information needed to resume the game.

2.2. Arithmetic

`/usr/games/arithmetic` [+–x/] [*range*]

`arithmetic` types out simple arithmetic problems, and waits for you to type an answer. If you answer correctly, `arithmetic` types back `Right!` and then a new problem. If the answer is wrong, it replies `What?` and waits for another answer. Every twenty problems, it publishes statistics on correctness and the time required to answer.

The first optional argument determines the kind of problem `arithmetic` poses:

+	for addition
–	for subtraction
x	(lowercase letter x) for multiplication
/	for division

You can combine these options. If you select more than one, `arithmetic` will mix the different types of problems in random order. The default is +–; in other words, addition and subtraction problems.

range is a decimal number; all addends, subtrahends, differences, multiplicands, divisors, and quotients are less than or equal to the value of *range*, the default of which is 10.

At the start, all numbers less than or equal to *range* are equally likely to appear. If you make a mistake, `arithmetic` will emphasize the numbers in the problem that you missed when it creates new problems for you.

As a matter of educational philosophy, the program will not give correct answers, since you *should* be able to calculate them. Thus the program provides drill for someone just past the first learning stage, and does not teach number facts as such. For almost all users, the relevant statistic should be time per problem, not the percentage of correct responses.

2.3. Banner

`/usr/games/banner [-w` *n* `]` *text*

`banner` prints a large, high quality banner on the standard output. If you omit the *text*, `banner` prompts for and reads one line of standard input. If you type the −w option, the output width decreases from 132 to the *n* you indicate, useful for printing on a narrow terminal. If you don't specify a value of *n*, *n* is 80.

Hard copy output is up to 132 columns wide, with no breaks between the pages, and may be quite long.

Bugs

Several ASCII characters are not defined, notably <, >, [,], ^, _, {, }, |, and ~.

Some characters produce odd output:

- & produces a cent sign (monetary value)
- `` and ` produce single open quotes
- ′ produces a single close quote

The −w option skips some rows and columns to reduce output width, so smaller *n*'s produce grainier output. Sometimes `banner` overlaps letters.

2.4. Bcd

`/usr/games/bcd` *text*

`bcd` converts the literal *text* into a form familiar to old-timers, a visual representation of a computer input card. If you omit the *text*, `bcd` prompts for and reads one line of standard input.

2.5. Boggle

`/usr/games/boggle [+] [++]`

This program sharpens your skills at Boggle (TM Parker Bros.).

If you invoke `boggle` without arguments, it generates a 4 by 4 Boggle grid of letters. The object of `boggle` is to find, within 3 minutes, as many words as possible in that 4 by 4 grid of letters. You can find words formed from any sequence of 3 or more adjacent letters in the grid. The letters may join horizontally, vertically, or diagonally. However, you can't use a position in the grid more than once within any one word. In competitive play among humans, each player gets credit for the words that no other player has found; when you play with the computer, you just compare the words you find with the ones it finds in its dictionary `/usr/games/bogdict`, which it generates from `/usr/dict/words`.

Enter your words separated by spaces, tabs, or newlines. A bell will ring with 2 minutes, 1 minute, 10 seconds, 2 seconds, and 1 second left in the game, and when time is up. When entering words, you may only erase within the current word and your line kill character is ignored.

Advanced players may wish to invoke the program with 1 or 2 +'s as the first argument:

+	allows you to use grid positions more than once in each word
++	causes `boggle` to consider a position as adjacent to itself as well as its (up to) 8 neighbors.

If you invoke the program with 4 arguments of 4 letters each, (in general, `boggle appl epie moth erhd`) it forms the obvious Boggle grid and lists all the words it can find both in the grid and in its dictionary.

For a graphics device version of the game, see `boggletool` in Section 4.1.

Bugs

You can't complete words that you started just before the expiration of time. You can't surrender before time is up, except by exiting the program entirely.

2.6. Chess

`/usr/games/chess`

`chess` is a computer program that plays class D chess. You can indicate moves with either standard (descriptive) notation or in algebraic notation. The symbol + specifies check; `o-o` and `o-o-o` specify castling. To play black, type `first`; to print the board, type a carriage return.

`chess` echoes each move in the appropriate notation followed by the program's reply.

For a graphics device version of `chess`, try `chesstool` in Section 4.2 .

Diagnostics

The most cryptic diagnostic is `eh?` which means that the input is syntactically incorrect.

Files

`/usr/games/lib/chess.book`	book of opening moves

Bugs

You can only promote pawns to queens.

2.7. Ching

`/usr/games/ching` [*hexagram*]

The *I Ching* or *Book of Changes* is an ancient oracle that the Chinese have used for centuries as a source of wisdom and advice.

The text of the *oracle* consists of sixty-four *hexagrams,* each symbolized by a particular arrangement of six straight (——) and broken (– –) lines. Traditionally, one determines the lines, or *changes*, by fixing a question firmly in mind, then throwing three coins (or yarrow stalks) six times. Coins have an even side, worth two, and an odd side, worth three; the line for a given roll comes from the total value of that roll:

6	gives `-o-`, a *moving* line
7	gives `---`
8	gives `- -`
9	gives `-x-`, a *moving* line

So, if you roll `687968`, you've constructed the hexagram:

```
-o-
- -
---
-x-
-o-
- -
```

When you look up a hexagram in the *Book of Changes*, you find two major sections: the **Judgement** relates specifically to the matter at hand (for example, "It furthers one to have somewhere to go"), while the **Image** describes the general attributes of the hexagram and how they apply to one's own life ("Thus the superior man makes himself strong and untiring").

When any of the lines has the value six or nine, it is a *moving* line; *moving* lines include a significant appended **Judgement**. Furthermore, the moving lines are inherently unstable and change into their opposites; a second hexagram (and thus an additional **Judgement**) appears.

Using an algorithm suggested by S. C. Johnson, the UNIX oracle reads a question from the standard input (up to an EOF) and hashes the individual characters in combination with the time of day, process id, and any other magic numbers which happen to be lying around the system. The resulting value seeds a random number generator that drives a simulated coin–toss divination. `ching` pipes the answer through `nroff` to format it before it appears as standard output.

If you wish to remain steadfast in the old traditions, you can enter the numerical value of a hexagram you toss (`687968` for example) as an argument to `ching`.

Files `/usr/games/lib/ching.d/*`

2.8. Fish

`/usr/games/fish`

`fish` plays *Go Fish*, a children's card game. The object is to accumulate *books* of 4 cards with the same face value. Players alternate turns, each selecting a card from their own hands to ask for all cards of that face value in the other player's hand. If one player asks the other player for one or more cards of that face value, and the other player has them, the first player claims the cards and makes another request. Eventually, the first player asks for a card which is not in the second player's hand, so the second player replies, ''GO FISH!'' The first player then draws a card from the 'pool' of undealt cards, and draws again if this is the card last requested from the other player.

When a player makes a *book*, all four cards of one face, either through drawing or requesting, the player lays down the cards in that book and no further action takes place with that face value.

To play the computer, simply make guesses by typing `a`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `10`, `j`, `q`, or `k` when asked. Typing RETURN gives you information about the size of the computer's hand and the pool, and tells you about its books. If you type `p` as a first guess, `fish` puts you into 'pro' level; the amateur level is fairly easy.

2.9. Fortune

`/usr/games/fortune [ – ] [ –wsla ] [` *file* `]`

`fortune` with no arguments prints out a random adage. Options include:

–w To wait after printing the message, for an amount of time calculated from its length. This is especially useful if one executes *fortune* as part of the logout procedure, so one can read the fortune before the screen clears.

–s Only short fortunes.

–l Only long fortunes.

–a Choose from either list of adages.

Files

`/usr/games/lib/fortunes.dat`

2.10. Monop

`/usr/games/monop [` *file* `]`

`monop` is reminiscent of the Parker Brother's game Monopoly, and monitors a game between 1 to 9 players. The game follows the standard rules, with the exception that, if a property would go up for auction and there are only two solvent players, no auction is held and the property remains unowned.

The game bank will lend players money, so it is possible to buy something that you cannot afford. However, players who are in debt must ''fix the problem,'' making themselves solvent, before play can continue. If a player cannot remain solvent, that player's property reverts to the player's debtee, either another player or the bank. A player can resign at any time to any person or the bank, which puts their property back on the board, unowned.

Any time that the response to a question is a *string* , in general, a name, place or person, you can type `?` to get a list of valid answers. It is not possible to input a negative number, nor is it ever necessary.

Summary of Commands

`quit`	Quit the game. It asks you if you're sure.
`print`	Prints out the current board. These columns appear on the board chart (column headings are the same for the `where`, `own holdings`, and `holdings` commands):

`Name`

The first ten characters of the name of the square

`Own`

The *number* of the owner of the property.

`Price`

The cost of the property (if any)

`Mg`

This field has a * in it if the property is mortgaged

#

For Utilities and Railroads, this is the number owned. If the property is land, this is the number of houses on it.

`Rent`

Current rent on the property. If no one owns it, there is no rent.

`where`	Tells you where all the players are. A * indicates the current player.
`own holdings`	List your own holdings, including money, get-out-of-jail-free cards, and property.
`holdings`	Look at anyone's holdings. It will ask you whose holdings you wish to look at. When you are finished, type `done`.
`shell`	Escape to a shell. When you terminate the shell, `monop` continues where you left off.
`mortgage`	Sets up a list of mortgageable property, and asks which you wish to mortgage.
`unmortgage`	Unmortgage mortgaged property.

`buy`	Sets up a list of monopolies on which you can buy houses. If you have more than one monopoly, `monop` asks you which you want to buy houses for, and how many houses you want to buy for each piece of property, giving the current amount in parentheses after the property name. If you don't build as evenly as possible on all of the properties in a monopoly, `monop` asks you to try again.
`sell`	Sets up a list of monopolies from which you can sell houses.
`card`	Use a get-out-of-jail-free card to get out of jail. If you're not in jail, or you don't have one, it tells you so.
`pay`	Pay $50 to get out of jail, from whence you are put on Just Visiting. Difficult to do if you're not there.
`trade`	Trade with another player. `monop` asks you with whom you wish to trade, and then asks each of you what you wish to trade. You can get a summary list and, in all cases, trading players must agree to confirm the trade before it takes place.
`resign`	Resign to another player or the bank. If you resign to the bank, all the property you owned reverts to the game bank, and get-out-of-jail free cards revert to the deck.
`save`	Save the current game in a file for later play. You can continue play after saving, either by adding the file in which you saved the game after the *monop* command, or by using the *restore* command (see below). It will ask you which file you wish to save it in, and, if the file exists, confirm that you wish to overwrite it.
`restore`	Read in a previously saved game from a file. It leaves the file intact.
`roll`	Roll the dice and move forward to your new location. You can simply type [RETURN] instead of typing `roll`.

Files

`/usr/games/lib/cards.pck`	Chance and Community Chest cards

Bugs

You can't specify command arguments to `monop` during runtime; instead, you must answer its inquiries.

2.11. Number

`/usr/games/number`

`number` copies the Arabic numeral standard input to the standard output, changing each decimal number to the fully spelled-out English version.

2.12. Quiz

`/usr/games/quiz [-i`*file* `] [-t ] [` *category1 category2* `]`

`quiz` tests your knowledge on various subjects. In an associative quiz, it asks items chosen from *category1* and expects answers from *category2*. If you don't specify the categories, `quiz` gives instructions and lists the available categories.

If you can't figure out a question, type a carriage return and `quiz` will provide the correct answer. At the end of input, when quitting, or when questions run out, `quiz` reports a score and terminates.

Options:

`-t` specifies 'tutorial' mode, where `quiz` repeats missed questions later, while continually introducing new material.

`-i` substitutes *file* for the default index file.

Substitute index files have this syntax:

```
line      = category newline | category ':' line
category  = alternate | category '|' alternate
alternate = empty | alternate primary
primary   = character | '[' category ']' | option
option    = '{' category '}'
```

The first category on each line of an index file names an information file. The remaining categories specify the order and contents of the data in each line of the information file. Information files have the same syntax. [Huh???] Use the backslash character, as with the shell, to quote syntactically significant characters or to insert transparent newlines into a line. `quiz` won't ask questions from index file lines that have empty question or answer entries.

Files

`/usr/games/quiz.k/*`

Bugs

The construct `a | ab` doesn't work in an information file; use `a{b}` instead.

2.13. Trek

`/usr/games/trek [ [ -a ] file ]`

`trek` is a game of space glory and war. In `trek`, you become the captain of the *U.S.S. Enterprise*, a starship that travels the galaxy attempting to destroy the Klingon starship fleet and save the Federation.

The *galaxy* consists of 64 *quadrants* on an eight by eight grid, with quadrant 0,0 in the upper left-hand corner. Each quadrant contains 100 *sectors*, in a ten by ten grid. Sectors may contain objects, like the Enterprise, Klingon starships, stars, or starbases, at which you can dock to refuel and to repair damages.

If you specify a *file* on the command line, `trek` writes a log of the game to that file. If you give the `-a` flag before the filename, `trek` appends the log to that file, rather than replacing it.

The game will ask you what length game you would like. You can respond: `short`, `medium`, or `long`. You may also type `restart` to continue playing a game you saved previously.

When `trek` asks you how skilled you are, you can respond: `novice`, `fair`, `good`, `expert`, `commodore`, or `impossible`. Start with novice and work up from there.

Throughout the game, if you need help type `?`.

Summary of Commands

`trek` requires direction (in degrees) and distance in quadrants (or quadrant fractions for sectors) to navigate through the galaxy. For example, to move up one quadrant, type:

```
move 0 1
```

or you can type `move`, after which `trek` will prompt you for the course, then for the distance.

You can use the bold-face portion of the command as its abbreviation.

`abandon`
Abandon ship.

`**ca**pture`
Request surrender of Klingon starship; if accepted, you can take captives to a Federation starbase for extra points.

`**cl**oak **u**p` or `**cl**oak **d**own`
When you `cloak up`, the Klingons cannot see you or fire on you. However, cloaking requires a lot of energy, so you can't fire weapons while cloaked. You must use a command that consumes time for the cloaking process to complete.

`**c**omputer request;` *request*; ...
Computer requests are:

`score`
Show your current score.

`course` *quad/sect*
Computes course and distance from your current position to the

quadrant *quad* and sector

you indicate.

`move` *quad/sect*
: Same as `course`, except `trek` completes the movement.

`trajectory`
: Prints the course and distance to all Klingons in the quadrant.

`warpcost` *dist warp_factor*
: Computes the cost in time and energy to move *dist* quadrants at warp *warp_factor*.

`impcost` *dist*
: Computes the cost in time and energy to move *dist* quadrants with impulse engines.

`pheff` *range*
: Tells how effective phasers are at range *range*.

`distresslist`
: Gives list of distressed starbases and quadrants, in other words, those occupied by Klingons.

You can make more than one request per command by seperating requests with semi-colons.

damages	Tells you which devices need repair; if you dock at a starbase, `trek` repairs them faster.
destruct	Self-destruct the starship, destroying any objects left in the quadrant.
dock	You can dock at a starbase when you are in one of the eight sectors adjacent to it. Starbases resupply you with energy, photon torpedoes, and life support reserves. They undertake any necessary repairs and protect you with their deflector shields. Unload prisoners at starbases to receive points for capturing them.
help	Don't use this command unless absolutely necessary; it counts heavily against you. The starbase you contact with `help` will try to rematerialize you at the starbase, but it doesn't always work.
impulse *course distance*	Move under impulse power, *course* in degrees, and *distance* in quadrants (for example, distance of `0.1` is one sector). `impulse` requires 20 energy units per move, as well as 10 energy units per sector moved. No penalty for shields. Klingons may attack you if you enter a quadrant they occupy. If the computer isn't working, you can't detect navigation errors, so you should return quickly to a starbase to get it fixed.
lrscan	Long range scan gives you a picture of your quadrants and the eight quadrants that surround it. Three-digit numbers tell you the number of stars (units digit), the number of starbases (tens digit), the number of Klingons (hundreds digit) in

	each quadrant. * indicates the impassible energy barrier at the end of the universe and /// marks a supernova quadrant you cannot enter.
move *course distance*	Move under warp power, *course* in degrees, and *distance* in quadrants (for example, distance of `0.1` is one sector). `move` consumes time (proportional to the inverse of the warp factor squared and directly proportional to the distance) and energy (proportional to the warp factor cubed and directly proportional to the distance). If you move with shields up, `move` consumes twice the amount of energy. Klingons may attack you if you enter a quadrant they occupy. If the computer isn't working, you can't detect navigation errors, so you should return quickly to a starbase to get it fixed.
phasers **a**utomatic *amount*	Shoot phasers on automatic with strength *amount*. 250 units of hits destroys a Klingon starship. Phaser effect decreases with distance; they have no effect outside the quadrant you occupy. You can't shoot phasers with your shields up, and they have no effect on starbases or stars.
phasers **m**anual *amt1 course1 spread1* ...	Shoot phasers manually with strength *amt1*, direction *course1*, and spread *spread1* (between 0 and 1.0). You may specify up to six manual phaser shots with one command. See `phaser automatic` for the rest of the details.
ram *course distance*	Identical to `move`, except that the computer won't stop you from navigation error. You suffer heavy consequences if you hit anything.
rest time	Allows you to rest to repair damages; inadvisable while under attack. Consumes time.
shell	Temporary escape to a shell. When you terminate the shell, you return to the game.
shields [up/down]	To raise or lower shields. Raising shields requires energy. Every time your shields, Klingons may attack you even though shields are only at partial effectiveness. Shields never provide complete protection from attack.
srscan [yes/no]	Short range scan gives you a picture of the quadrant you are in. It includes the following symbols: E the starship Enterprise K a Klingon starship # a starbase * a star . empty space a black hole
status	current status of the game

`terminate [yes/no]`	Cancels the current game. If you specify `yes`, `trek` will start a new game. If not, `trek` quits.
`torpedo` *course* `[yes/no]` *angle*	Fire photon torpedoes. If you hit a Klingon starship or a starbase, you destroy it. If you hit a star, it will go nova or supernova. Photon torpedoes are hard to aim, and although you can fire them with your shields up, the shields make them less accurate. Specify the direction with *course*, `no` if you don't want the three torpedoes in a burst fired at angles from one another, `yes` and *angle*, or just *angle*, if you want the torpedoes to burst at with an angle of degree *angle* (from `1` to `15`).
`undock`	Leave starbase.
`visual` *course*	For use when your short range scanners are broken, `visual` gives you a scan of the three sectors in direction `course` you specify. `visual` consumes 0.005 stardates.
`warp` *warp_factor*	Set warp factor. Minimum *warp_factor* is 1.0; maximum is 10.0. Probability of warp engine damage increases with warp factor increase over threshold of 6.0. Above warp 9.0, you may enter a time warp.

2.14. Wump

`/usr/games/wump`

wump plays the game of *Hunt the Wumpus*. A Wumpus is a creature that lives in a cave with several rooms connected by tunnels. You wander among the rooms, trying to shoot the Wumpus with an arrow, meanwhile avoiding being eaten by the Wumpus and falling into Bottomless Pits. There are also Super Bats which are likely to pick you up and drop you in some random room.

wump asks various questions which you answer one per line.

wump evolved from a program described in *People's Computer Company*, 2(2). November 1973.

3

Display Games

3

Display Games

Display games run on video terminals and graphics devices.

3.1. Backgammon

`backgammon [ - ] [ n r w b pr pw pb tterm sfile ]`

`backgammon` lets you play backgammon with the computer or with a friend.

Answer `yes` when `backgammon` asks if you want the rules, and you will get text explaining the rules of the game, some hints on strategy, instruction on how to use the `backgammon` program and a tutorial game with the computer. If you want to skip the rules and strategy section, to see the instructions only, answer `no` to the first inquiry, then `yes` when `backgammon` asks if you want instructions. `backgammon` will give you help when you type `?`.

Backgammon Command Line Options

`n`	don't ask for rules or instructions
`r`	player is red (implies `n`)
`w`	player is white (implies `n`)
`b`	two players, red and white (implies `n`)
`pr`	print the board before red's turn
`pw`	print the board before white's turn
`pb`	print the board before both players' turns
`t`*term*	terminal is type *term*, which uses `/etc/termcap`, otherwise uses the TERM environment variable.
`s`*file*	recovers previously saved game from *file*. You could also execute the saved file directly, that is, type the filename as a shell command line.

You can precede arguments by –. You can concatenate arguments, but not after `s` or `t`, because of the arbitrary strings that follow them. `backgammon` ignores any unrecognized arguments. A lone – displays a list of options.

If *term* has capabilities for direct cursor movement, `backgammon` 'fixes' the board after each move, so you need not reprint the board, unless the screen suffers some horrendous malady. Also, in this case, `backgammon` will ignore any `p*` options.

Commands

All commands are one letter, so don't type a carriage return, except at the end of a move.

When `backgammon` prompts with your game color, type a space or carriage return to roll, or:

`d`	to double
`p`	to print the board
`q`	to quit
`s`	to save the game for later

When `backgammon` prompts with `Move:`, type:

`p`	to print the board
`q`	to quit
`s`	to save the game

or a *move,* which is a sequence of

s–f move from *s* to *f*

s/r
move one piece on *s* the roll *r*

separated by commas or spaces and ending with a newline. Possible abbreviations are:

s–f1–f2
means *s–f1,f1–f2*

s/r1r2
means *s/r1,s/r2*

Use `b` for bar and `h` for home, or `0` or `25` as appropriate.

See Also

for a graphics-oriented version of `backgammon`, see section 4.3.

Files

`/usr/games/teachgammon`	rules and tutorial
`/etc/termcap`	terminal capabilities

Bugs

`backgammon`'s strategy needs work.

If you type DELETE, the program will *quit*, not delete the character you just typed.

3.2. Canfield

```
/usr/games/canfield
/usr/games/cfscores [-a ][ username ]
```

In `canfield`, there are several card locations: the *stock*, the *foundations*, the *talon*, and the *tableau*.

- You can build *tableau* cards on each other in descending face value and alternate colors. You can move an entire pile of tableau cards as a unit in building.
- You can place the top cards of the tableau piles on one of the four *foundations*, one for each suit in ascending face value, but never into empty spaces in the tableau.
- You may only fill tableau spaces from the top card of the *stock*.
- You can also use the top card of the stock to build foundations or tableau piles. If you exhaust the stock, you may fill tableau spaces from the *talon*, or the tableau spaces open until you wish to use them.

Type `ht` to get `canfield` to deal cards onto the talon. It deals cards from your (invisible) hand to the talon by threes; you can repeat this until your hand is empty, you quit, or you lose. `canfield` automatically moves foundation *base* cards to the foundation when they become available.

When you type `c`, `canfield` maintains card counting statistics on the bottom of the screen, greatly increasing your chances of winning, if you know how to use the information appropriately.

The betting rules are less strict than those used in the official version of the game. The initial deal costs $13. You may quit at this point or inspect the game. Inspection costs $13 and allows you to make as many moves as possible without moving any cards from your hand to the talon. (The initial deal places three cards on the talon; if you use all of these cards, `canfield` makes three more available to you.) Finally, if the game seems interesting, you must pay the final installment of $26. At this point, *canfield* credits you at the rate of $5 for each card on the foundation; and as the game progresses, you get $5 credit for each card moved to the foundation. Each additional run through the hand after the first run costs $5. The card-counting feature costs $1 for each unknown card you identify. If you choose to view the scoring information, `canfield` charges you only for cards that become visible since you last turned on the score-viewing option. Thus the maximum cost of information is $34. *canfield* charges for playing time at the rate of $1 per minute.

With no arguments, the program `cfscores` prints out the current status of your canfield account. If you specify a *username*, `cfscores` prints out the status of that `canfield` account. If you specify the –a option, `cfscores` prints out the `canfield` accounts for all users that have played the game since its database initialization.

Files

`/usr/games/lib/cfscores`	the database of scores

Bugs

It is impossible to cheat.

3.3. Cribbage

`/usr/games/cribbage [-req]` *name*

`cribbage` plays one hand of the card game cribbage, and you play the other. `cribbage` asks you if you need the rules of the game — if so, `cribbage` displays the appropriate section from *According to Hoyle*.

Options:

- `–e` Provides an explanation of the correct score when you make mistakes scoring your hand or crib. This is especially useful for beginners.
- `–q` Prints a shorter form of all messages — this is only recommended for users who know the game fairly well.
- `–r` Instead of asking you to cut the deck, `cribbage` cuts the deck for you.

Playing Cribbage

After asking you if you want instructions, `cribbage` asks you if you want to play a short game (''once around'' to 61) or a long game (''twice around'' to 121). Type `s` for a short game, or any other response for a long game.

At the start of the first game, `cribbage` asks you to cut the deck to determine who gets the first crib. Respond with a number between 0 and 51, indicating how many cards down the deck you want the cut. The player who cuts the lower ranked card gets the first crib. If you play more than one game, the loser of the previous game gets the first crib in the current game.

For each hand, `cribbage` prints the hand of the player with the crib. If you have the crib, `cribbage` will prompt you to discard two cards into the crib, one per line, as explained below.

After discarding, `cribbage` cuts the deck (when it is your crib) or asks you to cut the deck (when it is the program's crib). In the latter case, respond with a number from 0 to 39 indicating how far down in the deck you want the remaining 40 cards to be cut.

After cutting the deck, the person who doesn't have the crib leads the first card. Play continues until all of the cards are played. `cribbage` keeps track of the scoring of all points and the total of the cards on the table.

After play, `cribbage` computes the score for all of the hands. `cribbage` asks you to score your own hand (and the crib, if it is yours) by printing out the appropriate cards (and the cut card enclosed in brackets). Play continues until one player reaches the game limit (61 or 121).

If you type a carriage return when `cribbage` expects a numeric input, `cribbage` interprets this as typing the lowest legal value; therefore, when cutting the deck, a carriage return chooses the top card.

Specifying Cards

Specify cards as *rank* followed by *suit*. Specify the rank as one of: `a, 2, 3, 4, 5, 6, 7, 8, 9, t, j, q,` and `k`, or one of: `ace, two, three, four, five, six, seven, eight, nine, ten, jack, queen,` and `king`.

Specify the suit as one of: `s`, `h`, `d`, and `c`, or one of: `spades`, `hearts`, `diamonds`, and `clubs`.

Specify a card as *rank suit*, or *rank* of *suit*. If you use single-letter *rank* and *suit* designations, you can leave out the space separating the *suit* and *rank*. Also, when you can play only one card of the desired *rank*, typing only the *rank* of that card is sufficient. For example, if your hand is `2h`, `4d`, `5c`, `6h`, `jc`, and `kd`, and you want to discard the king of diamonds, you can type any of: `k`, `king`, `kd`, `k d`, `k of d`, `king d`, `king of d`, `k diamonds`, `k of diamonds`, `king diamonds`, or `king of diamonds`.

3.4. Hangman

`/usr/games/hangman`

In `hangman`, the computer picks a word from an on-line word list and you try to guess it. The computer keeps track of which letters you guess and how many wrong guesses you make.

Files

`/usr/dict/words` On-line word list

3.5. Mille

`/usr/games/mille` [*file*]

`mille` plays a two-handed game reminiscent of the Parker Brother's game of Mille Bornes.

When you start a game, the bottom of the score window will contain a list of commands. They are:

`P` Pick a card from the deck. This card is placed in the 'P' slot in your hand.

`D` Discard a card from your hand. To indicate which card, type the number of the card in the hand (or `P` for the just-picked card) followed by a carriage-return or space. (`mille` requires the carriage-return or space to allow recovery from drastic typos, like mistaken safety discards.)

`U` Use a card. Indicate the card by its number, followed by a carriage-return or space.

`O` Toggle ordering the hand. By default off, if turned on it will sort the cards in your hand appropriately. This is not recommended for the impatient on slow terminals.

`Q` Quit the game. This will ask for confirmation, just to be sure. Hitting [DELETE] (or [RUBOUT]) is equivalent.

`S` Save the game in a file. If you restarted the game from a file, you can save it again on the same file. If you don't wish to, or you did not start from a file, `mille` asks you for a filename in which to save the game. If you type a carriage-return without a name, `mille` will resume play without saving the game.

To resume a saved game, type:

`mille` *file*

on the command line, and the game you saved in that file will restart.

`R` Redraw the screen from scratch. The command CTRL-L will also work.

`W` Toggle window type. This switches the score window between the startup window (with all the command names) and the end-of-game window. Using the end-of-game window saves time by eliminating the switch at the end of the game to show the final score. Recommended for hackers.

If you make a mistake, `mille` prints an error message on the last line of the score window, and beeps.

At the end of each hand or game, `mille` will inquire if you wish to play another. If not, it will ask you if you want to save the game.

Cards

The number in brackets after the card name is the frequency of that card in the deck:

Hazard	Repair	Safety
Out of Gas [2]	Gasoline [6]	Extra Tank [1]
Flat Tire [2]	Spare Tire [6]	Puncture Proof [1]
Accident [2]	Repairs [6]	Driving Ace [1]
Stop [4]	Go [14]	Right of Way [1]
Speed Limit [3]	End of Limit [6]	

25 – [10], 50 – [10], 75 – [10], 100 – [12], 200 – [4]

Rules

Object: The point of game is to get a total of 5000 points in several hands. Each hand is a race to put down exactly 700 miles before your opponent does. Beyond the points gained by putting down milestones, there are several other ways of making points.

Overview: The game is played with a deck of 101 cards. *Distance* cards represent a number of miles traveled. They come in denominations of 25, 50, 75, 100, and 200. When you play a distance card, you add the card value to your trip mileage for that hand. Use *Hazard* cards to prevent your opponent from putting down *Distance* cards. With the exception of the *speed limit* card, your can only play the following cards if your opponent has a *Go* card on top of the Battle pile: *Out of Gas*, *Accident*, *Flat Tire*, *Speed Limit*, and *Stop*. *Remedy* cards fix problems caused by *Hazard* cards your opponent plays on you. These cards are: *Gasoline*, *Repairs*, *Spare Tire*, *End of Limit*, and *Go*. *Safety* cards prevent your opponent from putting specific Hazard cards on you in the first place. They are: *Extra Tank*, *Driving Ace*, *Puncture Proof*, and *Right of Way*, and there is only one of each in the deck.

Board Layout: The board splits into several areas. From top to bottom, they are: **SAFETY AREA** (unlabeled): Where safeties go when played. **HAND**: The cards in your hand. **BATTLE**: The Battle pile. You play Hazard and Remedy Cards here, except the *Speed Limit* and *End of Limit* cards. Only the top card is displayed, as it is the only effective one. SPEED: The Speed pile. You play *Speed Limit* and *End of Limit* cards here to control the speed at which the other player may put down miles. MILEAGE: Place *Mile* cards here. `mille` displays the total mileage here.

Play: First pick alternates between the two players. Usually, you start each turn with a pick from the deck. You then play a card, or if you can't or don't want to, you discard one. Normally, a play or discard of a single card constitutes a turn. If you play a safety card, however, you take another turn immediately.

This repeats until one of the players reaches 700 points or the deck runs out. If a you reach 700 points, you have the option trying an *Extension*, which means that the play continues until someone reaches 1000 miles.

Hazard and Remedy Cards

Play *Hazard* cards on your opponent's Battle and Speed piles. Use `Remedy` cards to undo the effects of your opponent's nastiness.

Go (Green Light) You must have the *Go* card on top of your Battle pile to play any mileage, unless you have played the *Right of Way* card (see below).

Stop
: Play this on your opponent's *Go* card to prevent your opponent from playing mileage cards until after playing a *Go* card.

Speed Limit
: Play this on your opponent's Speed pile. Then, until your opponent plays an *End of Limit*, your opponent can only play 25 or 50 mile cards, presuming the *Go* card allows them to do even that.

End of Limit
: Play this on your Speed pile to nullify a *Speed Limit* played by your opponent.

Out of Gas
: Play this on your opponent's *Go* card. Your opponent must then play a *Gasoline* card, and then a *Go* card before they can play any more mileage.

Flat Tire
: Play this on your opponent's *Go* card. Your opponent must then play a *Spare Tire* card, and then a *Go* card before playing any more mileage.

Accident
: Play this on your opponent's *Go* card. Your opponent must then play a *Repairs* card, and then a *Go* card before playing any more mileage.

Safety Cards

Safety cards prevent your opponent from playing the corresponding *Hazard* cards on you for the rest of the hand. You can cancel an attack in progress, and it *always entitles the player to an extra turn*.

Right of Way
: prevents your opponent from playing both *Stop* and *Speed Limit* cards on you. It also acts as a permanent *Go* card for the rest of the hand, so you can play mileage as long as there is not a Hazard card on top of your Battle pile.

In this case only, your opponent can play *Hazard* cards directly on a *Remedy* card besides a *Go* card.

Extra Tank
: When you play this card, your opponent cannot play an *Out of Gas* on your Battle Pile.

Puncture Proof
: When you play this card, your opponent cannot play a *Flat Tire* on your Battle Pile.

Driving Ace
: When you play this card, your opponent cannot play an *Accident* on your Battle Pile.

Distance Cards

Play *distance* cards when you have a *Go* card on your Battle pile, or a *Right of Way* in your Safety area and your opponent hasn't stopped you with a *Hazard* card. You can play *distance* cards in any combination that totals exactly 700 miles, except that *you cannot play more than two 200 mile cards in one hand*. A hand ends whenever one player gets exactly 700 miles or the deck runs out. If the deck runs out, you continue play until someone reaches 700, or neither player can play any cards. If you complete the trip after the deck runs out, this is called *Delayed Action*.

Coup Fouré

This is a French fencing term for a counter-thrust move as part of a parry to an opponents attack. In Mille Bornes, it is used as follows: If an opponent plays a *Hazard* card, and you have the corresponding *Safety* in your hand, you play it immediately, even *before* you draw. This immediately removes the *Hazard* card from your Battle pile, and protects you from that card for the rest of the game. This gives you more points (see **Scoring** below).

Scoring

`mille` totals scores at the end of each hand, whether or not anyone completed the trip. The Score window uses the following terms:

Milestones Played
: Each player scores as many miles as they played before the trip ended.

Each Safety
: 100 points for each safety in the Safety area.

All 4 Safeties
: 300 points if all four safeties are played.

Each Coup Fouré
: 300 points for each Coup Fouré accomplished.

The following bonus scores can apply only to the winning player.

Trip Completed
: 400 points bonus for completing the trip to 700 or 1000.

Safe Trip
: 300 points bonus for completing the trip without using any 200 mile cards.

Delayed Action
: 300 points bonus for finishing after the deck was exhausted.

Extension
: 200 points bonus for completing a 1000 mile trip.

Shut-Out
: 500 points bonus for completing the trip before your opponent played any mileage cards.

`mille` also keeps running totals of the current score for each player for the hand (*Hand Total*), the game (**Overall Total**), and number of games won (**Games**).

3.6. Snake

```
/usr/games/snake [-w n ] [-l n ]
/usr/games/snscore
```

`snake` is a chase game in which you try to make as much money as possible without getting eaten by the snake. The `-l` and `-w` options allow you to specify the length and width of the field. Unless you specify otherwise, `snake` uses the entire screen (except for the last column).

You show up on the screen as an `I`. The snake is 6 squares long and appears synaesthetically as a series of `S`'s. The money is `$`, and the exit is `#`. `snake` posts your score in the upper left-hand corner.

You can move around using the same conventions as `vi` — the `h`, `j`, `k`, and `l` keys work, as do the arrow keys. Other possibilities include:

`sefc`
: These keys are like `hjkl`, but form a directed pad around the `d` key.

`HJKL`
: These keys move you all the way in the indicated direction to the same row or column as the money. This does *not* let you jump away from the snake, but rather saves you from having to type a key repeatedly. The snake still slithers all of its turns.

`SEFC`
: Likewise for the upper case versions on the left side of the keyboard.

`ATPB`
: These keys move you to the four edges of the screen. Their position on the keyboard is the mnemonic — for example, `P` is at the far right of the keyboard.

`x` To quit the game.

`p` Points in a direction you might want to go.

`w` Space warp to get out of tight squeezes, at a price.

`!` Shell escape.

CTRL-Z Suspend the snake game, on systems which support it. Otherwise start an interactive shell.

To earn money, move to the same square the money is on. A new $ will appear when you earn the current one. As you get richer, the snake gets hungrier. To leave the game, move to the exit (#).

`snscore` keeps a record of the best score of each player. It counts your score only if you leave at the exit; if the snake eats you, you don't score.

As in pinball, if you match the last digit of your score to the number which appears after the game, you win a bonus.

To see who plays `snake`, and their scores, run `/usr/games/snscore`.

Files

/usr/games/lib/snakerawscores	database of player scores
/usr/games/lib/snake.log	log of games played

Bugs

When playing on a small screen, it's hard to tell when you hit the edge of the screen.

The scoring function takes into account the size of the screen. `snake` has not devised a perfect function to score equitably.

3.7. Worm

`/usr/games/worm` [*size*]

In `worm`, you are a little worm, your body is the `o`'s on the screen and your head is the `@`. You move with the `hjkl` keys (as in the game `snake`). If you don't press any keys, you continue in the direction you last moved. The upper-case `HJKL` keys move you as if you had pressed several of the corresponding lower-case key (9 for `HL` and 5 for `JK`).

On the screen you will see a *digit*. If your worm eats the digit, the worm will grow longer by the amount of the digit you ate. The object of the game is to see how long you can make the worm grow.

The game ends when the worm runs into either the sides of the screen, or itself. `worm` keeps the current score (how much the worm has grown) in the upper left corner of the screen.

The optional argument *size*, if present, is the initial length of the worm.

Bugs

If you set the initial length of the worm to less than one or more than 75, various strange things happen.

3.8. Worms

`/usr/games/worms` [–*field*] [– *length* #] [– *number* #] [– *trail*]

`worms` animates worms on terminal.

–field
makes a *field* for the worm(s) to eat;

–trail
causes each worm to leave a trail behind it.

`worms` evolved from a TOPS-20 program on the DEC-2136 machine called `worm`.

Files

`/etc/termcap`

See Also

Snails , by Karl Heuer

Bugs

The lower right-hand character position will not update properly on a terminal that wraps at the right margin.

`worms` will not initialize the terminal.

4

Graphics Games

4

Graphics Games

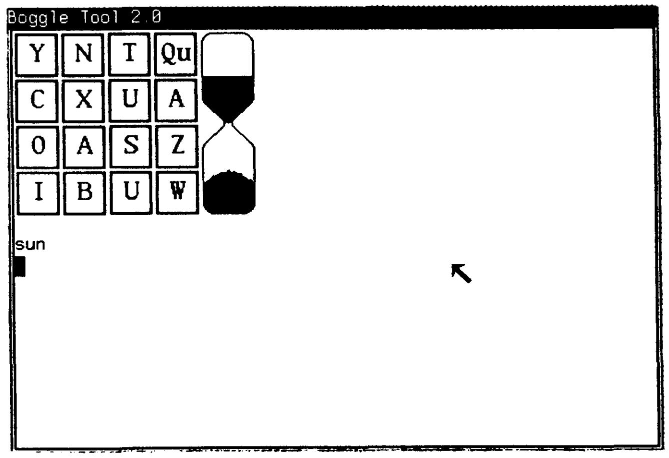
Graphics games will only run on devices that support graphics.

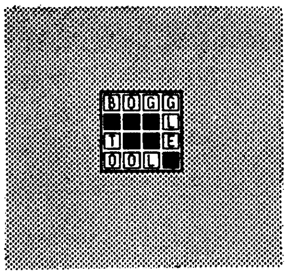
4.1. Boggletool

`boggletool` [*number*] [+ [+]] [*16-character string*]

`boggletool` allows you to play the game of Boggle (TM Parker Bros.) with the computer. The *number* argument specifies the time limit in minutes (the default is 3 minutes). If you put the *16-character string* on the command line, `boggle` interprets it as a Boggle board: the first four letters form the top row, the next four letters the second row, etc. If you don't specify any letters, `boggle` generates a ``random'' Boggle board. Explanation of the + [+] argument appears in the *Advanced Play* section.

Figure 4-1 `boggletool`





`boggletool` icon

Rules of the Game

The object of Boggle is to find as many words as possible in a 4 by 4 grid of letters within a certain time limit. You can form words from any sequence of 3 or more adjacent letters in the grid. The letters may join horizontally, vertically, or diagonally. Normally, you cannot use any letter in the grid more than once in a word (see *Advanced Play* for exceptions).

Playing the Game

When invoked, `boggletool` displays a grid of letters and an hourglass. To enter word guesses, simply spell the word you have found in lower-case letters. Use any white space (space, tab, or newline) to finish a word. To correct any mistakes, type the backspace key or [DEL] to delete the last character, or use [CTRL-U] to delete an entire word.

`boggletool` verifies that words you enter are both in the grid and are valid English words. If you try to type in a character that would form a word which is not in the grid, the display will flash and the character you typed will not be echoed. When you type any white space to end the current word, `boggletool` will verify that the word is three or more letters long and that it appears in the dictionary. If the word you typed is illegal for either reason, the display will flash and you will have to either erase the word or change it. If you try to reenter a valid word that you have already entered, the display will flash and the previous occurrence of the word will be highlighted. Again, you will have to erase the word before continuing.

As you enter words, the 'sand' in the hourglass will fall. At the end of the time limit, the display will flash and `boggletool` won't allow you to enter any more words. After a moment, the computer displays two lists of words: the words you found, and other words which also appear in the grid. To play another game, just type any capital letter (or use the pop-up menu).

Using the Menu

Access the pop-up menu by pressing the right button. The four items available work as follows:

`Restart Game`
: causes `boggletool` to create a new board, reset the timer, and allow you to start from scratch.

`Restart Timer`
: allows you to cheat by reseting the hourglass timer to zero.

`Give Up`
: causes `boggletool` to end the game and print the results immediately.

`Quit`
: allows you to quit `boggletool`. A prompt appears asking you to confirm the quit; when it does, click the left button to quit or the right button to abort the quit.

Advanced Play

There are two options for advanced players. If you type + as a command line option, `boggletool` allows you to reuse letters in the grid. If you type ++ as a command line option, `boggletool` considers letters to be adjacent to themselves as well as to their neighbors. Although it is far easier to find words with these two options, `boggletool` will find many more words in the grid, so it is more difficult to match the computer.

`boggletool` evolved from the terminal game `boggle` (Section 2.5).

Files

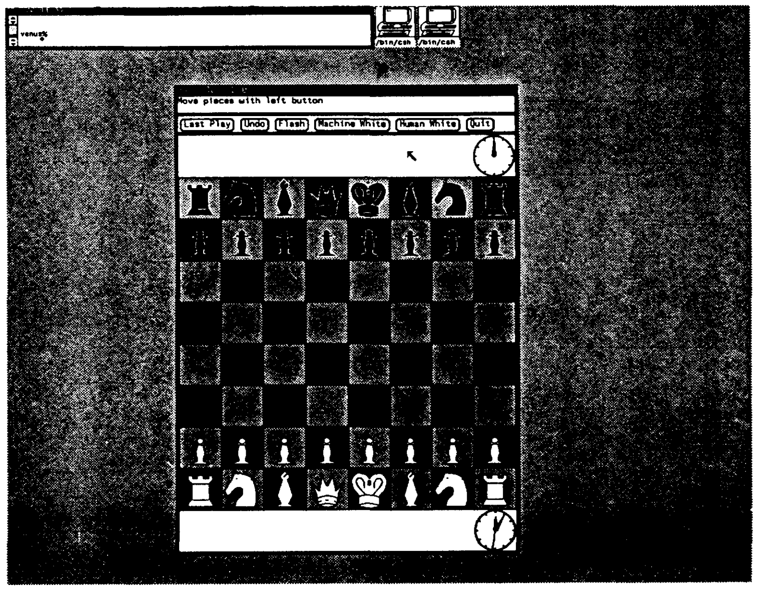
`/usr/games/boggledict`	dictionary file for computer's words

4.2. Chesstool

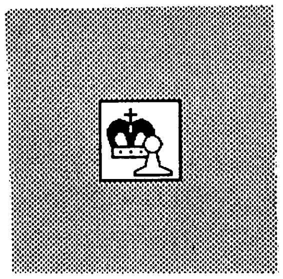
`chesstool` [*chess_program*]

`chesstool` is a graphics terminal version of the `chess` program (See Section 2.6). If you invoke `chesstool` without options, it uses `/usr/games/chess`; you can designate an alternate program which uses the same command syntax as `chess` with the *chess_program* argument.

Figure 4-2 `chesstool`



Move pieces with left button



`chesstool` icon

When you start `chesstool`, it displays a large window with three subwindows. The first subwindow displays messages — `Illegal move`, for example. The second subwindow is an options subwindow; options are described below. The final subwindow is a chessboard display with white and black pieces and two timekeeping clocks (advisory only).

Make your moves with the mouse: select a piece by positioning the arrow cursor over the piece and pressing the left mouse button down, then drag the piece to the destination square, and release the button. The cursor will then turn to an hour-glass icon while the system plays.

Select options in the options subwindow with either the left or middle mouse buttons. These options are:

`(Last Play)`	Show the last play made.
`(Undo)`	Undo your last move and the machine's response. Once the game is over, it is not possible to restart it, so `(Undo)` will update the board, but you cannot continue the game from that position.

`[Flash]`	Flash when the machine has completed its move. In flash mode, if the `chesstool` is open, the piece moved by the system on its play will flash until you make your move. If the `chesstool` is in icon form, the entire icon will flash when the machine has made its move. Thus you can `Close` the `chesstool` still know when it's your turn to move. To turn flash mode off, select `[Flash]` again.
`(Machine White)`	Start a new game with the machine playing white.
`(Human White)`	Start a new game with the machine playing black.

There are two special moves: castling and capturing a pawn *en passant.* To castle, move the king only. The position of the rook updates automatically. Since the king moves two squares when castling, the move is unambiguous. To capture *en passant,* move the pawn to the square occupied by the opposing pawn to capture it.

4.3. Gammontool

`gammontool` [*path*]

`gammontool` paints a board on a graphics tool device, then lets you play backgammon with the computer. The optional *path* argument specifies an alternate move-generating program, which must be compatible with `gammontool`.

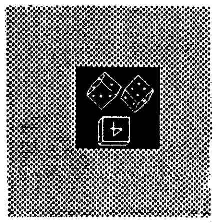
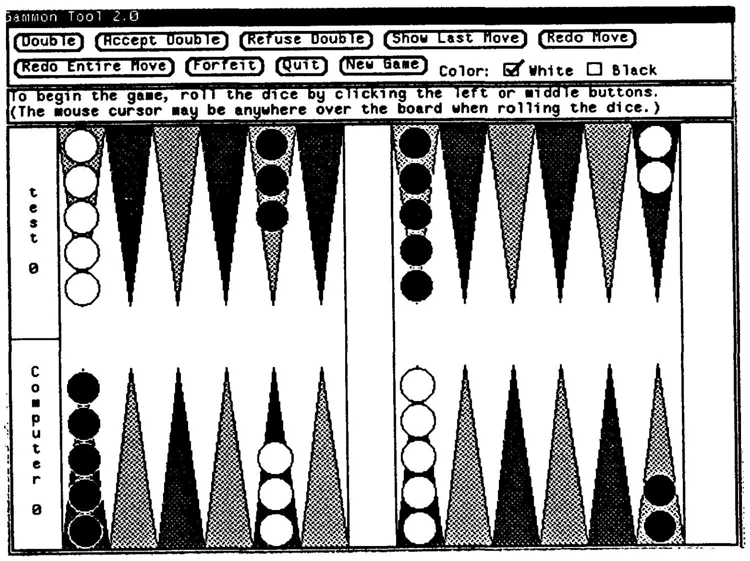

`gammontool` icon

Figure 4-3 `gammontool`



```
To begin the game, roll the dice by clicking the left or middle buttons.
(The mouse cursor may be anywhere over the board when rolling the dice.)
```

`gammontool` has three subwindows: an option window on top, a message window in the middle, and a large board on the bottom. Buttons in the option window restart, double, etc. The message window has two lines: the first tells whose turn it is, and the second displays any errors that occur.

The initial roll

To start the game, roll the dice to determine who goes first. Move the mouse arrow onto the board and click the left button. One die appears on each side of the board: the die on the left is yours, and the die on the right is the computer's. If your roll is greater, then you move; if not, the computer makes a move.

Making your move

When it is your turn, `Your turn to move` appears in the message window. Place the mouse over any piece of your color, and click the left button. While holding down the button, move the mouse to drag the piece; the piece follows the mouse until you release the button. The tool checks each move and does not allow illegal moves. When you have made as many moves as you can, the computer takes its turn; after it finishes, you may either roll again, or double.

Doubling

To double, click the `Double` button in the option window and wait for the computer's response. If the computer doubles you, it displays a message and you must answer with the `Accept Double` or `Refuse Double` buttons. You can also use the `Forfeit` button to refuse a double. If the game is doubled, a doubling cube with the proper value appears on the bar strip. If the number is face-up, then you may double next. If the number is upside-down, it is the computer's option to double.

Other buttons

If you want to change your move before you have finished it, use the `Redo Move` or `Redo Entire Move` buttons in the option window. `Redo Entire Move` replaces all of the pieces you have moved so that you can redo them all. `Redo Move` only replaces the last piece you moved, so it is useful when you roll doubles and want to redo only the last piece you moved. Note that once you have made all of the moves your roll permits, play passes immediately to the computer, so you cannot redo the very last move. The `Show Last Move` button allows you to see the last move again.

Leaving the game

If you want to quit playing backgammon, use the `Quit` button. If you want to forfeit the game, use the `Forfeit` button. The computer penalizes you by taking a certain number of points, but the program does not terminate.

To play another game after winning, losing, or forfeiting, click the `New Game` button. To change the color of your pieces, click the mouse button while pointing at either the `White` or `Black` checkboxes. You may change colors at any time, even in the middle of a game. Changing colors in the middle of a game does not mean that you trade places with the computer; your pieces stay where they are, but `gammontool` repaints them with the new color. Your pieces always move from the top right to the bottom right of the board, regardless of your color. As an additional cue to your color, `gammontool` always displays your dice on the left half of the board.

Log file

If a there is a `gammonlog` file in your home directory, `gammontool` keeps a log of the games you have played. It records each move and double, along with the winners and accumulated scores.

Files

`~/gammonlog` log of games played

Bugs

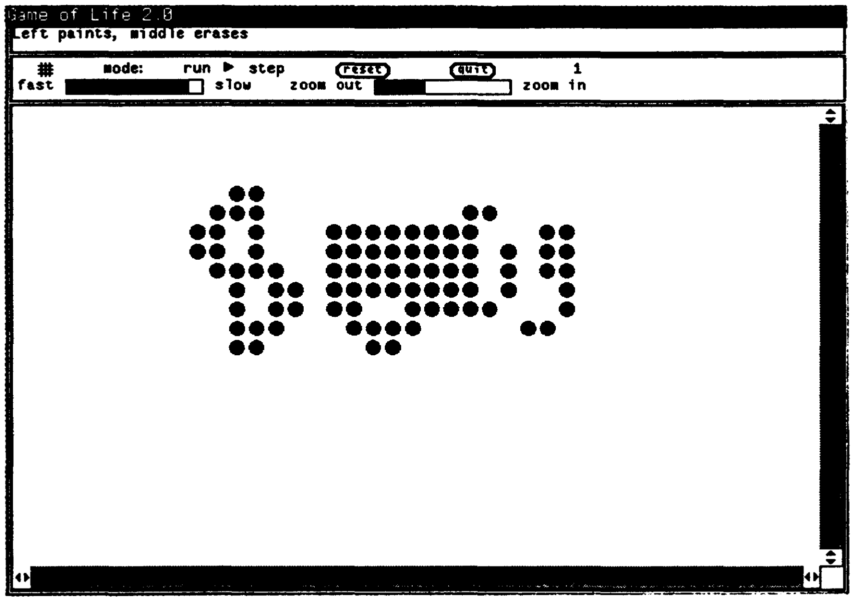
The computer's strategy is poor.

If a single move uses more than one die (for instance if you roll `5, 6` and move 11 spaces without touching down in the middle), `gammontool` may have difficulty deciding where to make the piece land. This may be important if there is a blot on one of the middle points. The program will always make the move if possible, but if two midpoints would work and there is a blot on one of them, it is much better to explicitly hit the blot, then move the piece the rest of the way.

4.4. Life

`life` is a program that plays John Conway's game of life.

Figure 4-4 `life`



`life` displays a window with a small control panel at the top, and a large drawing area at the bottom. You create pieces in the drawing area with the left button, and erase them with the middle button.

When you select `run` in the control panel, the pieces begin to evolve, and the drawing region updates itself at a speed controlled by the slider labeled with `fast` and `slow`.

`life` keeps track of all the pieces even if they are not visible. You can use the scroll bars surrounding the drawing region to see pieces that have moved out of view. You can draw some standard patterns by selecting from a menu that you can "pop up" in the drawing subwindow.

The first row of the control panel (from left to right) includes a variety of items:

the picture which looks like a tic-tac-toe board
  draws a grid in the drawing region

`mode`
: changes from *run* mode (where the pieces update continuously) to *step* mode (where an update occur only when you click on the `step` label)

`clear`
: clears the drawing region

`quit`
: exit from the game.

To the right of the `quit` button is a counter that records the generation number. The second row contains two sliders. The first controls the update speed when in run mode, the second controls the size of the pieces.

5

Graphics Standards Demos

5 Graphics Standards Demos

Graphics Standards demos comprise only the SunCore graphics standard at present.

5.1. Draw

`/usr/demo/draw`

The *draw* program is a menu-driven program that uses the mouse, keyboard, bit-map display and optionally the color display to draw objects, drag them around, save them on disk, and so on.

The main menu items are selected by moving the mouse cursor and pressing the left mouse button. To redraw the display, point at the left edge of the main menu box and press the left button. The main menu items are listed here.

New Seg xlate

Open a new translatable segment. A segment is a collection of attributes and primitives (lines, text, polygons, etc.). A translatable segment may subsequently be positioned.

New Seg xform

Open a new transformable segment. A transformable segment may subsequently be rotated, scaled, or positioned.

Delete Seg

To delete a segment, point at any primitive in the segment and press the left button.

Lines

To add line primitives to the currently open segment, position cursor, press the left button. Press right button to quit.

Polygon

To add a polygon primitive to the currently open segment, position the cursor, press the left button. Press the right button to terminate the boundary definition. Polygons are filled with the current fill attribute.

Raster

To add a raster primitive to the currently open segment, position the cursor, press the left button to reposition the box, adjust the box by moving the mouse, press the right button to create the raster primitive comprising the boxed bitmap. A 'rasterfile' is also created on disk for hardcopy purposes (see `/usr/include/rasterfile.h`). This 'rasterfile' file may be spooled to a Versatec printer/plotter for hardcopy after exiting from the draw program. The command to do this is:

```
lpr -v rasterfile
```

Text To add a text primitive to the currently open segment, position cursor, press left button, type the text string at the keyboard (back space works), hit return. Text is drawn with the current text attributes.

Marker To add marker primitives to the currently open segment, position cursor, press the left button to place marker. Press the right button to quit.

Position To position a segment, point at any primitive in the segment, press left button, position the segment, press right button to quit.

Rotate To rotate a transformable segment, point at any primitive in the segment, press left button, move mouse to rotate, press right button to quit.

Scale To scale a transformable segment, point at any primitive in the segment, press the left button, move mouse to scale in x or y, press right button to quit.

Attributes This item brings up the attribute menu. To select an attribute such as text font, region fill texture (color), linestyle, or line width, point at the item and press the left button. Point at the left edge of the menu box to quit.

Save Seg To save a segment on a disk file, point at the segment, press the left button, type the disk file name, hit return.

Restore Seg To restore a previously saved segment from disk, type file name, hit return.

Exit Exit the draw program.

Bugs

- Rasters and raster text do not scale or rotate.
- If segments completely overlap, only the last one drawn may be picked by pointing with the mouse. This also applies to the menu segments! Therefore, don't cover them up with polygons.
- If aborted with your interrupt character, you must type a `reset` command to turn keyboard echo back on and to reset *–cbreak*. Therefore, use the `Exit` item in the main menu to exit the program.

5.2. Suncube

`suncube` works on the color displays. It draws a multicolored Sun logo in the shape of a cube, then 'ripples' the color map through various shades.

6

PixRect Graphics Demos

6

PixRect Graphics Demos

Sun Pixrect Graphics Demos use the Pixrect layer of the graphics programming environment available on the Sun workstation.

6.1. Bouncedemo

`bouncedemo` draws a canvas on the workstation screen and shows a bouncing ball (actually a square) bouncing up and down.

6.2. Jumpdemo

`jumpdemo` illustrates the simulated jump to hyperspace from the 'Star Wars' series.

6.3. Molecule

`molecule` draws colored molecules on the screen.

6.4. Spheresdemo

`spheresdemo` draws a collection of overlapping, shaded spheres on the screen.

A

dc and bc — Desk Calculators

A

dc and bc — Desk Calculators

dc and bc are a pair of complementary interactive languages to provide calculation functions (desk calculators) at the keyboard.

bc is a language that accepts programs written in the familiar style of higher-level programming languages and compiles output which is interpreted by dc. Some dc commands described below are designed for use by bc and are not easy for a human user to manipulate.

A.1. DC — Interactive Desk Calculator

dc is an interactive desk calculator program implemented on the UNIX system to do arbitrary-precision integer arithmetic. dc works like a stacking calculator using reverse Polish notation. It has provision for manipulating scaled fixed-point numbers and for input and output in bases other than decimal.

Huge Numbers

The size of numbers that can be manipulated is limited only by available memory storage. On typical implementations of UNIX, the size of numbers that can be handled varies from several hundred digits on the smallest systems to several thousand on the largest.

Stack Operation

Numbers that are typed into dc are put on a push-down stack. dc commands work by taking the top number or two off the stack, performing the desired operation, and pushing the result on the stack. If an argument is given, input is taken from that file until its end, then from the standard input.

Description of dc Commands

Here we describe the dc commands that are intended for use by people. The additional commands that are intended to be invoked by compiled output are described in the detailed description.

Any number of commands are permitted on a line. Blanks and newline characters are ignored except within numbers and in places where a register name is expected.

The following constructions are recognized:

Push Number onto Stack

Typing a *number* is a command to mean that the value of *number* is pushed onto the main stack. A number is an unbroken string of the digits 0-9 and the capital letters A through F which are treated as digits with values 10 through 15, respectively, and possible a decimal point. The number may be preceded by an underscore to input a negative number.

Binary Operators

Binary operators operate on the top two values on the stack. The two entries are popped off the stack and the result is pushed on the stack in their place.

`+`	Add the two values on the top of the stack.
`-`	Subtract the two values on the top of the stack.
`*`	Multiply the two values on the top of the stack.
`/`	Divide the two values on the top of the stack. The result of a division is an integer truncated toward zero.
`%`	Remainder the two values on the top of the stack.
`^`	Exponentiate the two values on the top of the stack. An exponent must not have any digits after the decimal point.

See the detailed description below for the treatment of numbers with decimal points.

Comparison Operators

The top two elements of the stack are popped and compared according to the relational operators defined below. Register *x* is executed if they obey the stated relation. Exclamation point is negation.

<*x*

>*x*

=*x* equal

!<*x*

!>*x*

!=*x* not equal

Stack Operations

The operations noted below operate on the stack or between the stack and named registers.

`s`*x*	Pop the value from the top of the main stack and stores that value into a register named *x*, where *x* may be any single character. If the `s` is capitalized, *x* is treated as a stack and the value is pushed onto it. Any character, even blank or newline, is a valid register name.
`l`*x*	Push the value in register *x* onto the stack. The register *x* is not altered. If the `l` is capitalized, register *x* is treated as a stack and its top value is popped onto the main stack.

All registers start with empty value which is treated as a zero by the `l` command and is treated as an error by the `L` command.

`d`	The top value on the stack is duplicated.
`p`	Display the value on the top of the stack. The top value remains unchanged.
`f`	Display all values on the stack and in registers.
`x`	Treat the top element of the stack as a character string, remove it from the stack, and execute it as a string of `dc` commands.

[*character string*]
: Put the bracketed *character string* onto the top of the stack.

`q` Exit the program. If executing a string, pop the recursion level by two. If `q` is capitalized, pop the top value on the stack and pop the string execution level by that value.

`v` Replace the top element on the stack by its square root. The square root of an integer is truncated to an integer. For the treatment of numbers with decimal points, see the detailed description below.

`!` Interpret the rest of the line as a UNIX command. Control returns to `dc` when the UNIX command terminates.

`c` All values on the stack are popped; the stack becomes empty.

`i` The top value on the stack is popped and used as the number radix for further input. If `i` is capitalized, the value of the input base is pushed onto the stack. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 or greater than 16.

`o` The top value on the stack is popped and used as the number radix for further output. If `o` is capitalized, the value of the output base is pushed onto the stack.

`k` The top of the stack is popped, and that value is used as a scale factor that influences the number of decimal places that are maintained during multiplication, division, and exponentiation. The scale factor must be greater than or equal to zero and less than 100. If `k` is capitalized, the value of the scale factor is pushed onto the stack.

`z` The value of the stack level is pushed onto the stack.

`?` A line of input is taken from the input source (usually the console) and executed.

A.2. BC — Arbitrary-Precision Desk Calculator

`bc` is a language and a compiler for doing arbitrary-precision arithmetic on the UNIX system. The output of the compiler is interpreted and executed by a collection of routines which can input, output, and do arithmetic on indefinitely large integers and on scaled fixed-point numbers. These routines are themselves based on a dynamic storage allocator. Overflow does not occur until all available memory is exhausted.

The `bc` language has a complete control structure as well as immediate-mode operation. Functions can be defined and saved for later execution. A small collection of library functions is also available, including sin, cos, arctan, log, exponential, and Bessel functions of integer order.

Some of the uses of the `bc` compiler are:

- computation with large integers,
- computation accurate to many decimal places,
- conversion of numbers from one base to another base.

The bc compiler was written to make conveniently available a collection of routines (as described in the section on dc) which are capable of doing arithmetic on integers of arbitrary size. The compiler is by no means intended to provide a complete programming language — it is a minimal language facility.

There is a scaling provision that permits the use of decimal point notation. Provision is made for input and output in bases other than decimal. Numbers can be converted from decimal to octal by simply setting the output base to 8.

The actual limit on the number of digits that can be handled depends on the amount of storage available on the machine. Manipulation of numbers with many hundreds of digits is possible.

The syntax of bc has been deliberately selected to agree substantially with the C language. Those who are familiar with C will find few surprises in this language.

Simple Computations with Integers

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type in the line:

```
142857 + 285714
```

bc responds immediately with the line

```
428571
```

The operators –, *, /, %, and ^ can also be used; they indicate subtraction, multiplication, division, remaindering, and exponentiation, respectively. Division of integers produces an integer result truncated toward zero. Division by zero produces an error comment.

Any term in an expression may be prefixed by a minus sign to indicate that it is to be negated (the 'unary' minus sign). The expression

```
7+–3
```

is interpreted to mean that –3 is to be added to 7.

More complex expressions with several operators and with parentheses are interpreted just as in FORTRAN, with ^ having the greatest binding power, then * and % and /, and finally + and –. Contents of parentheses are evaluated before material outside the parentheses. Exponentiations are performed from right to left and the other operators from left to right. The two expressions

```
a^b^c  and  a^(b^c)
```

are equivalent, as are the two expressions

```
a*b*c  and  (a*b)*c
```

bc shares with FORTRAN and C the undesirable convention that

```
a/b*c  is equivalent to  (a/b)*c
```

Internal storage registers to hold numbers have single lower-case letter names. The value of an expression can be assigned to a register in the usual way. The statement

```
x = x + 3
```

has the effect of increasing by three the value of the contents of the register named x. When, as in this case, the outermost operator is an =, the assignment is performed but the result is not printed. Only 26 of these named storage registers are available.

There is a built-in square root function whose result is truncated to an integer (but see scaling below). The lines

```
x = sqrt(191) x
```

produce the printed result

```
13
```

Bases

There are special internal quantities, called 'ibase' and 'obase'. The contents of 'ibase', initially set to 10, determines the base used for interpreting numbers read in. For example, the lines:

```
ibase = 8
11
```

will produce the output line

```
9
```

and you are all set up to do octal to decimal conversions. Beware, however of trying to change the input base back to decimal by typing

```
ibase = 10
```

Because the number 10 is interpreted as octal, this statement will have no effect. For those who deal in hexadecimal notation, the characters A–F (upper-case only) are permitted in numbers (no matter what base is in effect) and are interpreted as digits having values 10–15 respectively. The statement:

```
ibase = A
```

will change you back to decimal input base no matter what the current input base is. Negative and large positive input bases are permitted but useless. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

The contents of 'obase', initially set to 10, are used as the base for output numbers. The lines

```
obase = 16 1000
```

will produce the output line

```
3E8
```

which is to be interpreted as a 3-digit hexadecimal number. Very large output bases are permitted, and they are sometimes useful. For example, large numbers can be output in groups of five digits by setting 'obase' to 100000. Strange (that is, 1, 0, or negative) output bases are handled appropriately.

Very large numbers are split across lines with 70 characters per line. Lines which are continued end with \. Decimal output conversion is practically instantaneous, but output of very large numbers (that is, more than 100 digits) with other bases is rather slow. Non-decimal output conversion of a one hundred-digit number takes about three seconds.

It is best to remember that 'ibase' and 'obase' have no effect whatever on the course of internal computation or on the evaluation of expressions, but only affect input and output conversion, respectively.

Scaling

A third special internal quantity called 'scale' is used to determine the scale of calculated quantities. Numbers may have up to 99 decimal digits after the decimal point. This fractional part is retained in further computations. We refer to the number of digits after the decimal point of a number as its scale.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules. For addition and subtraction, the scale of the result is the larger of the scales of the two operands. In this case, there is never any truncation of the result. For multiplications, the scale of the result is never less than the maximum of the two scales of the operands, never more than the sum of the scales of the operands and, subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity 'scale'. The scale of a quotient is the contents of the internal quantity 'scale'. The scale of a remainder is the sum of the scales of the quotient and the divisor. The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer. The scale of a square root is set to the maximum of the scale of the argument and the contents of 'scale'.

All of the internal operations are actually carried out in terms of integers, with digits being discarded when necessary. In every case where digits are discarded, truncation and not rounding is performed.

The contents of 'scale' must be no greater than 99 and no less than 0. It is initially set to 0. In case you need more than 99 fraction digits, you may arrange your own scaling.

The internal quantities 'scale', 'ibase', and 'obase' can be used in expressions just like other variables. The line:

```
scale = scale + 1
```

increases the value of 'scale' by one, and the line

```
scale
```

displays the current value of 'scale'.

The value of 'scale' retains its meaning as a number of decimal digits to be retained in internal computation even when 'ibase' or 'obase' is not equal to 10. The internal computations (which are still conducted in decimal, regardless of the bases) are performed to the specified number of decimal digits, never hexadecimal or octal or any other kind of digits.

Functions

The name of a function is a single lower-case letter. Function names are permitted to collide with simple variable names. Twenty-six different defined functions are permitted in addition to the twenty-six variable names. The line:

```
define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements, which make up the body of the function, ending with a right brace }. Return of control from a function occurs when a return statement is executed or when the end of the function is reached. The return statement can take either of the two forms

```
return
return(x)
```

In the first case, the value of the function is 0, and in the second, it is the value of the expression in parentheses.

Variables used in the function can be declared as automatic by a statement of the form

```
auto x,y,z
```

There can be only one 'auto' statement in a function and it must be the first statement in the definition. These automatic variables are allocated space and initialized to zero on entry to the function and thrown away on return. The values of any variables with the same names outside the function are not disturbed. Functions may be called recursively and the automatic variables at each level of call are protected. The parameters named in a function definition are treated in the same way as the automatic variables of that function with the single exception that they are given a value on entry to the function. An example of a function definition is

```
define a(x,y){
    auto z
    z = x*y
    return(z) }
```

The value of this function, when called, will be the product of its two arguments.

A function is called by the appearance of its name followed by a string of arguments enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments is used.

Functions with no arguments are defined and called using parentheses with nothing between them: `b()`.

If the function *a* above has been defined, then the line

```
a(7,3.14)
```

would display the result 21.98, and the line

```
x = a(a(3,4),5)
```

would assign the value 60 to the register *x*.

Subscripted Variables

A single lower-case letter variable name followed by an expression in brackets is called a subscripted variable (an array element). The variable name is called the array name and the expression in brackets is called the subscript. Only one-dimensional arrays are permitted. The names of arrays are permitted to collide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use. Subscripts must be greater than or equal to zero and less than or equal to 2047.

Subscripted variables may be freely used in expressions, in function calls, and in return statements.

An array name may be used as an argument to a function, or may be declared as automatic in a function definition by the use of empty brackets:

```
f(a[])
define f(a[])
auto a[]
```

When an array name is so used, the whole contents of the array are copied for the use of the function, and thrown away on exit from the function. Array names which refer to whole arrays cannot be used in any other contexts.

Control Statements

The `if`, the `while`, and the `for` statements may be used to alter the flow within programs or to iterate. The range of each of them is a statement or a compound statement consisting of a collection of statements enclosed in braces. They are written in the following way:

`if(` *relation* `)` *statement*
`while(` *relation* `)` *statement*
`for(` *expression-1* `;` *relation* `;` *expression-2* `)` *statement*

or

`if(` *relation* `)` { *statements* }
`while(` *relation* `)` { *statements* }
`for(` *expression-1* `;` *relation* `;` *expression-2* `)` { *statements* }

A *relation* in one of the control statements is an expression of the form:

```
x>y
```

where two expressions are related by one of the six relational operators <, >, <=, >=, ==, or !=. The relation == stands for 'equal to' and != stands for 'not equal to'. The meaning of the remaining relational operators is clear.

BEWARE of using = instead of == in a relational. Unfortunately, both of them are legal, so you will not get a diagnostic message, but = really will not do a comparison.

The `if` statement executes its range if and only if the relation is true. Then control passes to the next statement in sequence.

The `while` statement executes its range repeatedly as long as the relation is true. The relation is tested before each execution of its range and if the relation is false, control passes to the next statement beyond the range of the while.

The `for` statement begins by executing *expression1*. Then the *relation* is tested and, if true, the statements in the range of the `for` are executed. Then *expression2* is executed. The *relation* is tested, and so on. The typical use of the `for` statement is for a controlled iteration, as in the statement:

```
for(i=1; i<=10; i=i+1) i
```

which prints the integers from 1 to 10. Here are some examples of the use of the control statements.

```
define f(n){
auto i, x
x=1
for(i=1; i<=n; i=i+1) x=x*i
return(x)
}
```

The line:

```
    f(a)
```

prints *a* factorial if *a* is a positive integer. Here is the definition of a function which will compute values of the binomial coefficient (m and n are assumed to be positive integers).

```
define b(n,m){
auto x, j
x=1
for(j=1; j<=m; j=j+1) x=x*(n-j+1)/j
return(x)
}
```

The following function computes values of the exponential function by summing the appropriate series without regard for possible truncation errors:

```
scale = 20
define e(x){
    auto a, b, c, d, n
    a = 1
    b = 1
    c = 1
    d = 0
    n = 1
    while(1==1){
        a = a*x
        b = b*n
        c = c + a/b
        n = n + 1
        if(c==d) return(c)
        d = c
    }
}
```

Some Details

There are some language features that every user should know about even if he will not use them.

Normally statements are typed one to a line. It is also permissible to type several statements on a line separated by semicolons.

If an assignment statement is parenthesized, it then has a value and it can be used anywhere that an expression can. For example, the line

```
(x=y+17)
```

not only makes the indicated assignment, but also prints the resulting value.

Here is an example of a use of the value of an assignment statement even when it is not parenthesized.

```
x = a[i=i+1]
```

assigns a value to *x* and also increments *i* before it is used as a subscript.

The following constructs work in bc in exactly the same manner as they do in the C language. Consult any reference on the C language for their exact workings.

```
x=y=z  is the same as   x=(y=z)
x =+ y                  x = x+y
x =- y                  x = x-y
x =* y                  x = x*y
x =/ y                  x = x/y
x =% y                  x = x%y
x =^ y                  x = x^y
x++                     (x=x+1)-1
x--                     (x=x-1)+1
++x                     x = x+1
--x                     x = x-1
```

Even if you don't intend to use these constructs, if you type one inadvertently, something correct but unexpected may happen.

Note: In some of these constructions, spaces are significant. There is a real difference between x=-y and x= -y. The first replaces x by x-y and the second by -y.

Three Important Things

1. To exit a bc program, type quit.
2. There is a comment convention identical to that of C and of PL/I. Comments begin with /* and end with */.
3. There is a library of math functions which may be obtained by typing at command level:

   ```
   bc -l
   ```

   This command loads a set of library functions which, at the time of writing, consists of sine (named 's'), cosine ('c'), arctangent ('a'), natural logarithm ('l'), exponential ('e') and Bessel functions of integer order ('j(n,x)'). The library sets the scale to 20. You can reset it to something else if you like.

If you type

```
bc file ...
```

bc will read and execute the named file or files before accepting commands from the keyboard. In this way, you may load your favorite programs and function definitions.

Syntactic Description of bc

In the following pages syntactic categories are in *italics*; literals are in **boldface**; material in brackets [ ] is optional.

Tokens

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs or comments. Newline characters or semicolons separate statements.

Comments

Comments are introduced by the characters /* and terminated by */.

Identifiers

There are three kinds of identifiers — ordinary identifiers, array identifiers and function identifiers. All three types consist of single lower-case letters. Array identifiers are followed by square brackets, possibly enclosing an expression describing a subscript. Arrays are one-dimensional and may contain up to 2048 elements. Indexing begins at zero, so an array may be indexed from 0 to 2047. Subscripts are truncated to integers. Function identifiers are followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict; a program can have a variable named **x**, an array named **x** and a function named **x**, all of which are separate and distinct.

Keywords

The following are reserved keywords:

```
ibase   if     obase   break
scale   define sqrt    auto
length  return while   quit    for
```

Constants

Constants consist of arbitrarily long numbers with an optional decimal point. The hexadecimal digits A-F are also recognized as digits with values 10–15, respectively.

Expressions

The value of an expression is printed unless the main operator is an assignment. Precedence is the same as the order of presentation here, with highest appearing first. Left or right associativity, where applicable, is discussed with each operator.

Primitive expressions

Named expressions are places where values are stored. Simply stated, named expressions are lega! on the left side of an assignment. The value of a named expression is the value stored in the place named.

Simple identifiers are named expressions. They have an initial value of zero.

Array elements are named expressions. They have an initial value of zero.

The internal registers **scale, ibase** and **obase** are all named expressions. **scale** is the number of digits after the decimal point to be retained in arithmetic operations. **scale** has an initial value of zero. **ibase** and **obase** are the input and output number radix, respectively. Both **ibase** and **obase** have initial values of 10.

Function Calls

function-name ([*expression* [, *expression* . . .]])

A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value. As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a return statement, the value of the function is the value of the expression in the parentheses of the return statement or is zero if no expression is provided or if there is no return statement.

`sqrt` (*expression*)

The result is the square root of the expression. The result is truncated in the least significant decimal place. The scale of the result is the scale of the expression or the value of **scale**, whichever is larger.

`length` (*expression*)

The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

`scale` (*expression*)

The result is the scale of the expression. The scale of the result is zero.

Constants

Constants are primitive expressions.

Parentheses

An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

Unary operators

The unary operators bind right to left.

– *expression*

The result is the negative of the expression.

++ *named-expression*

The named expression is incremented by one. The result is the value of the named expression after incrementing.

— *named-expression*

The named expression is decremented by one. The result is the value of the named expression after decrementing.

named-expression ++

The named expression is incremented by one. The result is the value of the named expression before incrementing.

named-expression ––

The named expression is decremented by one. The result is the value of the named expression before decrementing.

Binary Operators

The exponentiation operator binds right to left.

expression ^ *expression*

The result is the first expression raised to the power of the second expression. The second expression must be an integer. If *a* is the scale of the left expression and *b* is the absolute value of the right expression, then the scale of the result is:

min ($a \times b$, max (**scale**, *a*))

The operators *, /, % bind left to right.

expression * *expression*

The result is the product of the two expressions. If *a* and *b* are the scales of the two expressions, the scale of the result is:

min ($a+b$, max (**scale**, *a*, *b*))

expression / *expression*

The result is the quotient of the two expressions. The scale of the result is the value of **scale**.

expression % *expression*

The % operator produces the remainder of the division of the two expressions. More precisely, *a%b* is *a–a/b*b*.

The scale of the result is the sum of the scale of the divisor and the value of **scale**

The additive operators bind left to right.

expression + *expression*

The result is the sum of the two expressions. The scale of the result is the maximun of the scales of the expressions.

expression – *expression*

The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

The assignment operators bind right to left.

named-expression = *expression*

This expression results in assigning the value of the expression on the right to the named expression on the left.

named-expression =+ *expression*
named-expression =− *expression*
named-expression =* *expression*
named-expression =/ *expression*
named-expression =% *expression*
named-expression =^ *expression*

The result of the above expressions is equivalent to ''named expression = named expression OP expression'', where OP is the operator after the = sign.

Unlike all other operators, the relational operators are only valid as the object of an **if**, **while**, or inside a **for** statement.

expression < *expression*
expression > *expression*
expression <= *expression*
expression >= *expression*
expression == *expression*
expression != *expression*

Storage classes

There are only two storage classes in bc, global and automatic (local). Only identifiers that are to be local to a function need be declared with the **auto** command. The arguments to a function are local to the function. All other identifiers are assumed to be global and available to all functions. All identifiers, global and local, have initial values of zero. Identifiers declared as **auto** are allocated on entry to the function and released on returning from the function. They therefore do not retain values between function calls. **auto** arrays are specified by the array name followed by empty square brackets.

Automatic variables in bc do not work in exactly the same way as in either C or PL/I. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

Statements

Statements must be separated by semicolon or newline. Except where altered by control statements, execution is sequential.

Expression statements

When a statement is an expression, unless the main operator is an assignment, the value of the expression is printed, followed by a newline character.

Compound statements

Statements may be grouped together and used when one statement is expected by surrounding them with { }.

Quoted string statements

```
"any string"
```

This statement prints the string inside the quotes.

`if` statements

`if` (*relation*) *statement*

statement is executed if the *relation* is true.

`while` statements

`while (` *relation* `)` *statement*

statement is executed while the *relation* is true. The test occurs before each execution of *statement*.

`for` statements

`for (` *expression*`;` *relation*`;` *expression* `)` *statement*

The `for` statement is the same as

```
first-expression
while (relation) {
      statement
      last-expression
}
```

All three expressions must be present.

`break` statements

`break` terminates a `for` or `while` statement.

Auto statements

`auto` *identifier* `[,` *identifier* `]`

`auto` pushes down the values of the identifiers. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. `auto` must be the first statement in a function definition.

`define` statements

`define(` `[`*parameter* `[ ,` *parameter* `. . . ] ] )` `{`*statements* `}`

`define` defines a function. The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

`return` statements

`return`

`return (` *expression* `)`

`return` terminates a function, pops its auto variables, and specifies the result of the function. The first form is equivalent to `return(0)`. The result of the function is the result of the expression in parentheses.

`quit`

`quit` stops execution of a `bc` program and returns control to UNIX when it is first encountered. Because `quit` is not treated as an executable statement, it cannot be used in a function definition or in an `if`, `for`, or `while` statement.

Index

Revision History

Version	Date	Comments
A	17 February 1986	Reworking of Section 6 Man Pages with additional descriptive text and organization.

Notes