NAME

bs − a compiler/interpreter for modest-sized programs

SYNOPSIS

bs [file [arg ...]]

DESCRIPTION

Bs is a remote descendant of Basic and Snobol4 with a little C language thrown in. *Bs* is designed for programming tasks where program development time is as important as the resulting speed of execution. Formalities of data declaration and file/process manipulation are minimized. Line-at-a-time debugging, the *trace* and *dump* statements, and useful run-time error messages all simplify program testing. Furthermore, incomplete programs can be debugged; *inner* functions can be tested before *outer* functions have been written and vice versa.

If the command line *file* argument is provided, the file is used for input before the console is read. By default, statements read from the file argument are compiled for later execution. Likewise, statements entered from the console are normally executed immediately (see *compile* and *execute* below). The result of an immediate expression statement is printed.

Bs programs are made up of input lines. If the last character on a line is the \, the line is continued. *Bs* accepts lines of the following form:

```
statement
label statement
```

A label is a *name* (see below) followed by a colon. A label and a variable can have the same name.

A *bs* statement is either an expression or a keyword followed by zero or more expressions. Some keywords (*clear*, *compile*, *!*, *execute*, and *run*) are always executed as they are compiled.

Statement Syntax:

expression
: The expression is executed for its side effects (value, assignment or function call). The details of expressions follow the description of statement types below.

break
: *Break* exits from the inner-most *for/while* loop.

clear
: Clears the symbol table and compiled statements. *Clear* is executed immediately.

compile [expression]
: Succeeding statements are compiled (overrides the immediate execution default). The optional expression is evaluated and used as a file name for further input. A *clear* is associated with this latter case. *Compile* is executed immediately.

include expression
: The expression should evaluate to a file name. The file must contain *bs* source statements. *Include* statements may not be nested.

continue
: *Continue* transfers to the loop-continuation of the current *for/while* loop.

dump
: The name and current value of every non-local variable is printed. After an error or interrupt, the number of the last statement and (possibly) the user-function trace are displayed.

exit [expression]
: Return to system level. The expression is returned as process status.

execute
Change to immediate execution mode (an interrupt has a similar effect). This statement does not cause stored statements to execute (see *run* below).

for name = expression expression statement
for name = expression expression
...
next

for expression , expression , expression statement
for expression , expression , expression
...
next
The *for* statement repetitively executes a statement (first form) or a group of statements (second form) under control of a named variable. The variable takes on the value of the first expression, then is incremented by one on each loop, not to exceed the value of the second expression. The third and fourth forms require three expressions separated by commas. The first of these is the initialization, the second is the test (true to continue), and the third is the loop-continuation action (normally an increment).

fun f(a [, ...]) [v, ...]
...
nuf
Fun defines the function name, arguments, and local variables for a user-written function. Up to ten arguments and local variables are allowed. Such names cannot be arrays, nor can they be I/O associated. Function definitions may not be nested.

freturn
A way to signal the failure of a user-written function. See the interrogation operator (?) below. If interrogation is not present, *freturn* merely returns zero. When interrogation *is* active, *freturn* transfers to that expression (possibly by-passing intermediate function returns).

goto name
Control is passed to the internally stored statement with the matching label.

if expression statement
if expression
...
[**else**
...]
fi
The statement (first form) or group of statements (second form) is executed if the expression evaluates to non-zero. The strings **0** and "" (null) evaluate as zero. In the second form, an optional *else* allows for a group of statements to be executed when the first group is not. The only statement permitted on the same line with an *else* is an *if*; only other *fi*'s can be on the same line with a *fi*.

return [expression]
The expression is evaluated and the result is passed back as the value of a function call. If no expression is given, zero is returned.

onintr label
onintr
The *onintr* command provides program control of interrupts. In the first form, control will pass to the label given, just as if a *goto* had been executed at the time *onintr* was executed. The effect of the statement is cleared after each interrupt. In the second form, an interrupt

will cause *bs* to terminate.

run
The random number generator is reset. Control is passed to the first internal statement. If the *run* statement is contained in a file, it should be the last statement.

stop
Execution of internal statements is stopped. *Bs* reverts to immediate mode.

trace [expression]
The *trace* statement controls function tracing. If the expression is null (or evaluates to zero), tracing is turned off. Otherwise, a record of user-function calls/returns will be printed. Each *return* decrements the *trace* expression value.

while expression statement
while expression
. . .
next
While is similar to *for* except that only the conditional expression for loop-continuation is given.

! shell command
An immediate escape to the Shell.

. . .
This statement is ignored. It is used to interject commentary in a program.

Expression Syntax:

name
A name is used to specify a variable. Names are composed of a letter (upper or lower case) optionally followed by letters and digits. Only the first six characters of a name are significant. Except for names declared in *fun* statements, all names are global to the program. Names can take on numeric (double float) values, string values, or can be associated with input/output (see the built-in function *open()* below).

name ([expression [, expression] . . .])
Functions can be called by a name followed by the arguments in parentheses separated by commas. Except for built-in functions (listed below), the name must be defined with a *fun* statement. Arguments to functions are passed by value.

name [expression [, expression] . . .]
Each expression is truncated to an integer and used as a specifier for the name. The resulting array reference is syntactically identical to a name. **a[1,2]** is the same as **a[1][2]**. The truncated expressions are restricted to values between 0 and 32767.

number
A number is used to represent a constant value. A number is written in Fortran style, and contains digits, an optional decimal point, and possibly a scale factor consisting of an e followed by a possibly signed exponent.

string
Character strings are delimited by " characters. The \ escape character allows the double quote (\"), new-line (\n), carriage return (\r), backspace (\b), and tab (\t) characters to appear in a string. Otherwise, \ stands for itself.

(expression)
Parentheses are used to alter normal order of evaluation.

(expression, expression [, expression . . .]) [expression]
The bracketed expression is used as a subscript to select a comma-separated expression from

the parenthesized list. List elements are numbered from the left, starting at zero. The expression

(False, True)[a == b]

has the value *True* if the comparison is true.

? expression
The interrogation operator tests for the success of the expression rather than its value. At the moment, it is useful for testing end-of-file (see examples in the *Programming Tips* section below), the result of the *eval* built-in function, and for checking the return from user-written functions (see *freturn*). An interrogation "trap" (end-of-file, etc.) causes an immediate transfer to the most recent interrogation, possibly skipping assignment statements or intervening function levels.

− expression
The result is the negation of the expression.

++ name
Increments the value of the variable (or array reference). The result is the new value.

−− name
Decrements the value of the variable. The result is the new value.

! expression
The logical negation of the expression. Watch out for the shell escape command.

expression *operator* expression
Common functions of two arguments are abbreviated by the two arguments separated by an operator denoting the function. Except for the assignment, concatenation, and relational operators, both operands are converted to numeric form before the function is applied.

Binary Operators (in increasing precedence):

=
= is the assignment operator. The left operand must be a name or an array element. The result is the right operand. Assignment binds right to left, all other operators bind left to right.

_
_ (underscore) is the concatenation operator.

& |
& (logical and) has result zero if either of its arguments are zero. It has result one if both of its arguments are non-zero. | (logical or) has result zero if both of its arguments are zero. It has result one if either of its arguments is non-zero. Both operators treat a null string as a zero.

< <= > >= == !=
The relational operators (< less than, <= less than or equal, > greater than, >= greater than or equal, == equal to, != not equal to) return one if their arguments are in the specified relation. They return zero otherwise. Relational operators at the same level extend as follows: *a>b>c* is the same as *a>b & b>c*. A string comparison is made if both operands are strings.

+ −
Add and subtract.

* / %
Multiply, divide, and remainder.

^

Exponentiation.

Built-in Functions:

Dealing with arguments

arg(i)
is the value of the *i*-th actual parameter on the current level of function call. At level zero, *arg* returns the *i*-th command argument (*arg(0)* returns **bs**).

narg()
returns the number of arguments passed. At level zero, the command argument count is returned.

Mathematical

abs(x)
is the absolute value of *x*.

atan(x)
is the arctangent of *x*. Its value is between $-\pi/2$ and $\pi/2$.

ceil(x)
returns the smallest integer not less than *x*.

cos(x)
is the cosine of *x* (radians).

exp(x)
is the exponential function of *x*.

floor(x)
returns the largest integer not greater than *x*.

log(x)
is the natural logarithm of *x*.

rand()
is a uniformly distributed random number between zero and one.

sin(x)
is the sine of *x* (radians).

sqrt(x)
is the square root of *x*.

String operations

size(s)
the size (length in bytes) of *s* is returned.

format(f, a)
returns the formatted value of *a*. *F* is assumed to be a format specification in the style of *printf*(3S). Only the %...f, %...e, and %...s
types are safe.

index(x, y)
returns the number of the first position in *x* that any of the characters from *y* matches. No match yields zero.

trans(s, f, t)
Translates characters of the source *s* from matching characters in *f* to a character in the

same position in *t*. Source characters that do not appear in *f* are copied to the result. If the string *f* is longer than *t*, source characters that match in the excess portion of *f* do not appear in the result.

substr(s, start, width)
returns the sub-string of *s* defined by the *start*ing position and *width*.

match(string, pattern)
mstring(n)
The *pattern* is similar to the regular expression syntax of the *ed*(1) command. The characters ., [,], ^ (inside brackets), * and $ are special. The *mstring* function returns the *n*-th (1 <= *n* <= 10) substring of the subject that occurred between pairs of the pattern symbols \(and \) for the most recent call to *match*. To succeed, patterns must match the beginning of the string (as if all patterns began with ^). The function returns the number of characters matched. For example:

```
match("a123ab123", ".*\([a−z]\)") == 6
mstring(1) == "b"
```

File handling

open(name, file, function)
close(name)
The *name* argument must be a *bs* variable name (passed as a string). For the *open*, the *file* argument may be **1)** a 0 (zero), 1, or 2 representing standard input, output, or error output, respectively, **2)** a string representing a file name, or **3)** a string beginning with an ! representing a command to be executed (via *sh −c*). The *function* argument must be either **r** (read), **w** (write), **W** (write without new-line), or **a** (append). After a *close*, the *name* reverts to being an ordinary variable. The initial associations are:

```
open("get", 0, "r")
open("put", 1, "w")
open("puterr", 2, "w")
```

Examples are given in the following section.

access(s, m)
executes *access*(2).

ftype(s)
returns a single character file type indication: **f** for regular file, **d** for directory, **b** for block special, or **c** for character special.

Odds and Ends

eval(s)
The string argument is evaluated as a *bs* expression. The function is handy for converting numeric strings to numeric internal form. *Eval* can also be used as a crude form of indirection as in

```
name = "xyz"
eval("++"_ name)
```

which increments the variable *xyz*. In addition, *eval* preceded by the interrogation operator permits the user to control *bs* error conditions. For example,

```
?eval("open(\"X\", \"XXX\", \"r\")")
```

returns the value zero if there is no file named "XXX" (instead of halting the user's program). The following executes a *goto* to the label *L* (if it exists).

```
label="L"
if !(?eval("goto "_ label)) puterr = "no label"
```

plot(request, args)

The *plot* function produces output on devices recognized by *plot*(1G). The *requests* are as follows.

Call	*Function*
plot(0, term)	causes further *plot* output to be piped into *plot*(1G) with an argument of -T*term*.
plot(1)	"erases" the plotter.
plot(2, string)	labels the current point with *string*.
plot(3, x1, y1, x2, y2)	draws the line between (*x1*,*y1*) and (*x2*,*y2*).
plot(4, x, y, r)	draws a circle with center (*x*,*y*) and radius *r*.
plot(5, x1, y1, x2, y2, x3, y3)	draws an arc (counterclockwise) with center (*x1*,*y1*) and endpoints (*x2*,*y2*) and (*x3*,*y3*).
plot(6)	is not implemented.
plot(7, x, y)	makes the current point (*x*,*y*).
plot(8, x, y)	draws a line from the current point to (*x*,*y*).
plot(9, x, y)	draws a point at (*x*,*y*).
plot(10, string)	sets the line mode to *string*.
plot(11, x1, y1, x2, y2)	makes (*x1*,*y1*) the lower right corner of the plotting area and (*x2*,*y2*) the upper left corner of the plotting area.
plot(12, x1, y1, x2, y2)	causes subsequent x (y) coordinates to be multiplied by *x1* (*y1*) and then added to *x2* (*y2*) before they are plotted. The initial scaling is plot(12, 1.0, 1.0, 0.0, 0.0).

Some requests do not apply to all plotters. All requests except zero and twelve are implemented by piping characters to *plot*(1G). See *plot*(5) for more details.

last()

in immediate mode, *last* returns the most recently computed value.

PROGRAMMING TIPS

Using *bs* as a calculator:

```
$ bs
# distance (inches) light travels in a nanosecond
186000 * 5280 * 12 / 1e9
11.78496
...
# Compound interest (6% for 5 years on $1000)
int = .06 / 4
bal = 1000
for i = 1 5*4  bal = bal + bal*int
bal − 1000
346.855007
...
```

```
        exit
```

The outline of a typical *bs* program:

```
        # Initialize things:
        var1 = 1
        open("read", "infile", "r")
        . . .
        # Compute:
        while ?(str = read)
                . . .
        next
        # Clean up:
        close("read")
        . . .
        # Last statement executed (exit or stop):
        exit
        # Last input line:
        run
```

Input/Output examples:

```
1)      #   Copy "oldfile" to "newfile".
        open("read", "oldfile", "r")
        open("write", "newfile", "w")
        . . .
        while ?(write = read)
        . . .
        #   Close "read" and "write"
        close("read")
        close("write")
2)      #   Pipe between commands
        open("ls", "!ls *", "r")
        open("pr", "!pr −2 −h 'List'", "w")
        while ?(pr = ls) . . .
        . . .
        #   Be sure to close (wait for) these
        close("ls")
        close("pr")
```

SEE ALSO

ed(1), plot(1G), sh(1), access(2), printf(3S), stdio(3S), Section 3 of this volume for further description of the mathematical functions (pow(3M) is used for exponentiation), plot(5). *Bs* uses the Standard Input/Output package.

BUGS

There are built-in design limits. *Bs* source programs are restricted to fewer than 250 lines and fewer than 250 variables (the *name* of an array counts as a variable, as does each dimension and each referenced element).

All names (labels, variables, functions, statement keywords) are internally truncated to six characters.