1. Introduction

`JaM` is a simple stack-based interactive system with graphics utilities. It is implemented in Mesa for the Xerox Alto and D-series computers. `JaM` is intended to be a exible system, giving the user rather direct control over the basic primitives. It is not intended to be a fault tolerant system for inexperienced users. This manual is written in the same spirit: the goal is to elaborate only those aspects of `JaM` which are not properties of programming languages in general. It provides explanations of all but the most obscure intrinsic functions and a sampling of the most useful external utilities.

For our purposes `JaM` has three major components: a virtual memory of 2^{24} words; a set of primitive objects and operations which use a stack discipline, much like a very powerful, exible Hewlett-Packard calculator; and a graphics package. The virtual memory is implemented by the `JaM` system software using a le called `JaM.VM.` To speed execution, objects are put in a special cache after being looked up in the `JaM.VM` le. Repeated calls to the same function can then be satis ed from the cache.

It is possible to use the virtual memory for long term storage of programs and data, although this is probably not a good idea. If one gets into serious trouble, it is also nice to be able to start over with a new `JaM.VM` le without losing all of one's work. For simplicity, virtual memory is always allocated sequentially from the end of the `JaM.VM` le with no garbage collection. This means the le keeps getting longer and longer. It is therefore necessary to ush the le periodically. In general, however, it takes a long time before this is necessary.

All functions in `JaM` make use of the operand stack. This stack contains objects: integers (long and short), reals, booleans, character strings, commands, streams (which are really les), dictionaries, arrays, and special stack markers. Execution proceeds by getting a token from the input stream, converting it into an object, checking whether or not it is executable and then executing it. Non-executable objects are pushed onto the stack. During execution, operands may be retrieved from the operand stack and results are returned on this stack. This means that all input is given in *post x* notation.

The graphics package is a group of functions, some written in `JaM`, others written directly in Mesa, which must be loaded in addition to the basic `JaM` system. Normally, these are loaded automatically, but when working with a brand new `JaM.VM` le they must be loaded by hand or by running a special initialization program. It is a good idea to do this, since `JaM` requires a good deal of initialization.

2. Basic Operation

`JaM` execution proceeds simply by transforming the input stream into a sequence of objects and processing them in order. There are two types of objects: *nouns* which are automatically placed on the operand stack, and *verbs* which are executed immediately. The `JaM` scanner parses the input stream into a series of tokens separated by tabs, carriage returns, spaces, or commas. There are three types of tokens: numbers, strings, and identi ers. Numbers and strings are converted directly into the corresponding objects and placed on the operand stack. Identi ers are looked up in the current dictionary (explained later) and the resulting object is processed according to whether it is a noun or a verb.

Syntactically, a string is a sequence of up to 2^{15} 1 characters enclosed in parentheses. It may contain anything except unbalanced parentheses (even carriage returns). Any token which is neither a number nor a string is an identifier.

The system has three major stacks: the operand stack, the dictionary or context stack, and the execution stack. The dictionary stack is used for keeping track of identifiers during execution. This stack functions as a set of nested contexts, much like the blocks of a block structured language. A dictionary is essentially a table of fixed size for associating identifiers with their values. When the scanner comes across an identifier, it tries to look it up in the dictionary on the top of the dictionary stack. If the lookup is successful, the value found in the dictionary is the next object returned by the scanner. Otherwise, the identifier is looked up in each of the other dictionaries on the dictionary stack in sequence, until a value is found.

The execution stack is used for keeping track of nested function calls. It contains the information necessary to implement recursive function invocations. It need not be the direct concern of the user.

3. How to Start

To run `JaM`, you need the following files: `JaM.bcd`, `CedarGraphics.bcd`, `Splines.bcd` and `JaMGraphics.bcd`. `JaM.bcd` contains the compiled Mesa program which runs the `JaM` system and the other three files implement most of the graphics routines. If you do not have a `JaM.VM` file, you should run `start.jam` to put some needed definitions in your virtual memory (see `.run`’’ under **Input/Output Commands**). This program requires more files: `util.jam`, `errordefs.jam`, `graphics.jam`, and `jamsave.jam`. If your environment does not include a lot of the standard Mesa bcd’s it is probably best to use a packaged up version of `JaM` available as `JaM.run`.

Once you have the files, type `jam`’’ to the Alto executive. This puts you in `JaM` you will get a prompt `*`’’, followed by a blinking cursor. The upper part of the screen is reserved for text. The lower part is used by the graphics utilites. Initially, the operand stack is empty and the dictionary stack contains only one dictionary, the system dictionary. All the `JaM` intrinsic functions are defined in this dictionary. There are a number of other functions which `JaM` assumes exist and expects the user to define. These will be explained later when we discuss the relevant details of the system. Running `start.jam` will define these functions (effectively give them default definitions) and it will load a number of useful utilities. The system dictionary is not big enough to hold all these definitions, so they are put in a new dictionary on the top of the dictionary stack. From now on, everything you need will be loaded automatically every time you use `JaM`.

4. Commands and Utilities

This discussion is organized by function. All commands relating to a particular topic are given together along with an explanation of the particular aspect of the system to which they relate. The exact placement on the stack of arguments and results for each function is given by a diagram. For example:

.add . . . .   <x><y> / <x + y>   n

The symbols in angle brackets represent objects on the stack; the rightmost object corresponds to the top of the stack. An arrow (/) separates the condition of the operand stack before the command is executed from the condition after. Only the top-most objects on the stack are shown, the others are assumed to remain unchanged. The symbol >will be used to mean the bottom of the stack. A comment explaining the command more fully usually follows the symbol n. There are various error conditions which can occur when trying to execute a command. These are discussed in a separate section (see **Error Handling**) because there are special functions relating to this topic.

5. Arithmetic and Bit Manipulation

There are three types of numeric objects: reals, integers, and long integers. Integers are 16 bits long; reals and long integers are 32 bits. Implicit type conversion is performed on numeric objects, but it is also possible to do these conversions explicitly (see **Type Conversion**).

.add . . . . <x><y> / <x + y> n

.sub . . . . <x><y> / <x y> n

.mul . . . . <x><y> / <xy> n

.div . . . . <x><y> / <x/y> n

.neg . . . . <x> / < x> n

.cos . . . . <x> / <cos(x)> n (x in degrees)

.sin . . . . <x> / <sin(x)> n

.atan . . . . <y><x> / <p = atan(y/x)> n (180 < p 180)

.exp . . . . <b><e> / <b^e> n (result always of real type)

.log . . . . <b><v> / <log_b v> n

.bitor . . . . <x><y> / <p>
n (The integer p is the bitwise or of 16 bit integers x and y.)

.bitxor . . . . <x><m > / <p>
n (p is the 16 bit exclusive or of x and y.)

.bitand . . . . <x><n> / <p> n (p is the 16 bit logical and of x and y.)

.bitnot . . . . <x> / <p> n (p is the 16 bit bitwise complement of x.)

.bitshift . . . . <x><y> / <p>
n (The integer x is shifted left by y places. If y > 0 then the y low order bits are set to zero. If y < 0 then the y high order bits are set to zero and y is shifted right. If jyj 16 then p is set to zero.)

6. Boolean and Relational Commands

The following commands deal with boolean objects. They have two possible values represented here by .true'' and .false''. For numeric comparisons, implicit type conversion occurs before comparison. If one argument is integer and the other is long integer or real, the integer is converted to that type. Similarly, long integers may be converted to real type. Strings may also be compared using lexicographic ordering. It is not legal to compare integers with strings.

.true . . . . / <.true> n

.false . . . . / <.false> n

`.eq` . . . . <x><y> / <`.true`> if x = y, otherwise <`.false`> n

`.gt` . . . . <x><y> / <`.true`> if x > y numerically or lexicographically, else <`.false`> n

`.lt` . . . . <x><y> / <`.true`> if x < y numerically or lexicographically, else <`.false`> n

`.not` . . . . <x> / < x> n

`.and` . . . . <x><y> / <x ^ y> n

`.or` . . . . <x><y> / <x _ y> n

`.xor` . . . . <x><y> / <x y> n

7. Stack Manipulation Commands

It is often useful to manipulate the operand stack. It is probably not good practice, however, to try to use the stack for all temporary storage. De ne local variables instead (see **Dictionary Related Commands**). Overuse of stack manipulation makes programs hard to read and di cult to debug. With this warning in mind, the following diagrams should make these commands clear.

`.pop` . . . . <x> / n

`.dup` . . . . <x> / <x><x> n

`.exch` . . . . <x><y> / <y><x> n

`.copy` . . . . <x_1><x_2>...<x_i><i> / <x_1><x_2>...<x_i><x_1><x_2>...<x_i> n (i must be of type integer)

`.roll` . . . . <x_1><x_2>...<x_i><i><j> / <$x_{(1 \ j)(\mathrm{mod}\ i)}$>...<$x_i$><$x_1$>... <$x_{\ j(\mathrm{mod}\ i)}$> n Here i and j are of type integer, and j may be negative. If j is postive then the top j elements of the stack are interchanged with the following i j, else the bottom jjj elements among the top i are brought to the top by a similar interchange. (Since we started counting from 1, not 0, by k (mod i) we mean i, when i divides k.)

`.index` . . . . <x_n><$x_{n\ 1}$>...<x_0><i> / <x_n><$x_{n\ 1}$>...<x_0><x_i> n (i n)

`.cntstk` . . . . ><x_1><x_2>...<x_i> / ><x_1><x_2>...<x_i><i> n

`.clrstk` . . . . ><x_1><x_2>...<x_i> / > n clears the stack

`/clr` . . . . ><x_1><x_2>...<x_i> / > n (This is a synonym for the intrinsic function `.clrstk`. It comes from the `util.jam` utility package.)

8. Stack Marking and Mark Manipulation Commands

There is a special object called a stack mark. Its main purpose is for keeping variable numbers of arguments on the stack. The `.loop` and `.rept` commands (see below) use this concept internally to allow execution inside the loop without losing track of the original condition of the execution stack.

`.mark` . . . . / <mark> n (mark type object put on stack)

`.cnttomrk` . . . . <mark><x$_1$><x$_2$>...<x$_i$> / <mark><x$_1$><x$_2$>...<x$_i$><i> n

`.clrtomrk` . . . . <mark><x$_1$><x$_2$>...<x$_i$> / <mark> n

9. Execution Control Commands

`JaM` has much of the execution control machinery found in Algol-like languages. This includes looping'' and if-else'' constructions. There is no go to'' command, however, as this would not easily t into the stack oriented structure of `JaM`. The constructions just mentioned do t in with the stack oriented structure because they can operate on executable objects in the operand stack. For example, the `.if` command expects a boolean object and any other object on the operand stack. If the boolean equals `.true`, then the other object is executed, otherwise the `.if` command pops the object from the stack.

`.exec` . . . . <x> / *see comment*
n Remove `x` from the operand stack and execute it as if it just came from the input stream. To reverse this e ect, see **Type Conversion** .

`.if` . . . . <b><x> / if `b` = `.true` then execute `x` n

`/if` . . . . <b><x> / *see comment*
n utility equivalent to: `.cvx .if`''

`.ifelse` . . . . <b><x><y> / if `b` = `.true` then execute `x` else execute `y` n

`/ifelse` . . . . <b><x><y> / *see comment*
n utility equivalent to: `.ifelse .cvx .exec`''

`.rept` . . . . <i><x> / execute `x` `i` times n

`.loop` . . . . <x> / execute `x` forever n

`.for` . . . . <i><j><k><x> / *see comment*
n Execute `x` b(k i)/jc+ 1 times, with `i` on top of the stack the rst time and `i+ j,i+ 2j,...` on top thereafter.

`.exit` . . . . / *see comment*
n Exit from current `.rept`, `.loop`, ..for `.dictforall`, or `.arrayforall` loop. This clears the execution stack down to the mark placed upon entering the current loop.

`.interrupt` . . . . / *see comment* n This function is called when the user presses the interrupt'' (right shift + SWAT) key. This is normally used for getting programs out of in nite loops. It prints `interrupt--`'', clears the stack, and executes a `.stop` command. You may rede ne this, but be careful!

`/pause` . . . . `/` *see comment*
n This simple utility function merely waits for the user to type something.

`.stop` . . . . `/` *see comment*
n Clears the execution stack, terminating all un nished execution. To see how this a ects error conditions, refer to **Error Handling** .

`.singlestep` . . . . `/` *see comment* n Puts execution in single step mode. The function `.step` is called each time a command is executed. You must de ne `.step` before trying to use this feature! The function `.step` might print whatever values you are interested in. You may want to make the printing conditional, or perhaps just gather statistics.

`.runfree` . . . . `/` *see comment*
n Take execution control out of single step mode.

`.quit` . . . . `/` *see comment*
n Save virtual memory and exit `JaM` gracefully.

10. Dictionary Related Commands

Variables are stored in special objects called dictionaries. Dictionary objects are general symbol tables useful for all kinds o storage. They have a xed maximum size speci ed at the time of their creation.

As mentioned earlier, there is a stack of dictionaries maintained by the system. The dictionaries in this stack are used like levels of static nesting for variable de nitions in an Algol-like language. Dictionaries may be named and retrieved for later use just like other objects in `JaM`.

There are a few things to watch out for when dealing with dictionaries. Some of the system commands and utilities assume that there is space left in the current dictionary (the one at the top of the dictionary stack) to de ne temporary variables. For this reason, you should be careful about creating small dictionaries. Another problem is what to do when dictionaries get lled up. There are four choices: push another dictionary onto the dictionary stack with `.begin`, remove the o ending dictionary with `.end`, delete entries to make room, or clear the dictionary. This problem is particularly severe for users who just start de ning variables without ever worrying about dictionaries. The system dictionary has a capacity of `256` entries, most of which are already lled up with system command de nitions. When the system dictionary gets lled up, whatever you do, don't clear it!

In the following table, <`k`> refer to keys or variables and <`v`> refers to the corresponding values.

`.curdict` . . . . `/` <current dictionary>
n Pushes the top of the dictionary stack onto the operand stack.

`.sysdict` . . . . `/` <system dictionary> n

`.dict` . . . . <`i`> `/` <`d`>
n Creates new dictionary `d` with capacity of `i` entries.

`.def` . . . . <`k`><`v`> `/` *see comment* n Associate the value `v` with the key k in the current dictionary. To de ne a function, make v executable.

`/def` . . . . <k><s><v> / *see comment*
n Equivalent to: <k> <v> `.def $help` <k> <s> `.put`’’. The `$help` dictionary contains informative messages about certain functions. It provides a convenient way to document programs. To access this dictionary, see `Input/Output Commands`. This utility resides in the le `util.jam`

`/xdef` . . . . <k><s><v> / *see comment*
n Equivalent to: `.cvx /def`’’.

`.del` . . . . <d><k> / *see comment*
n deletes the key k from the dictionary d

`.load` . . . . <k> / <v>
n Retrieve the value associated with the key k in the current context (i.e., dictionary stack). If v is an executable string, this allows its de nition to be printed. (See **Type Conversion** and **Input/Output Commands**).

`.store` . . . . <k><v> / *see comment* n Finds a de nition of k in the current context and replaces that de nition with the value v. If no de nition of k exists, this functions as `.def`.

`.put` . . . . <d><k><v> / *see comment*
n associates value v with key k in dictionary d

`.get` . . . . <d><k> / <v>
n retrieves the value v associated with k in dictionary d

`.known` . . . . <d><k> / *see comment*
n <`.true`> if key k is in dictionary d, <`.false`> otherwise.

`.where` . . . . <k> / *see comment* n <d><`.true`> if k is found in some dictionary d on the dictionary stack, <`.false`> otherwise.

`/dir` . . . . <d> / *see comment*
n Utility from `util.jam` which prints all key, value pairs in the dictionary d.

`/kdir` . . . . <d> / *see comment*
n Utility which prints all the keys in the dictionary d.

`??` . . . . <d> / *see comment* n This is a utility from `util.jam` equivalent to `$help /kdir`. It prints all functions on which help is available. This usually includes most of the external utility functions which have been loaded.

`.clrdict` . . . . <d> / *see comment* n Clear all entries from dictionary d

`.dictforall` . . . . <d><x> / *see comment*
n Put <k><v> on the stack, and then execute <x>. This is done for every k, v pair in dictionary d

`.begin` . . . . <d> / *see comment*
n Push d on the top of the dictionary stack. (This makes d the current dictionary.)

`.end` . . . . .end / *see comment*
n Pop current dictionary o the top of the dictionary stack.

`.length` . . . . <d> / <i>
n The current number of entries in the dictionary d.

`.maxlength` . . . . <d> / <I> n The total capacity of dictionary d.

11. Array Related Commands

Array objects are linear arrays of objects indexed starting at *zero*. They have xed lengths determined when they are created. There are commands for creating arrays, storing into them, retrieving objects from them, etc. Most commands either expect array objects on the operand stack or return array objects. Arrays can also be made executable. (See `Type Conversion`)

`.array` . . . . <i> / <a> n Creates a new array a of i objects.

`[` . . . . / <mark>
n Mark the operand stack for use by the]'' utility.

`]` . . . . / <a>
n Put all the objects down to the rst stack mark into an array a and return it on the operand stack. [obj$_0$ obj$_1$ ...obj$_{i\ 1}$]'' forms an i-element array.

`.subarray` . . . . <a><i><j> / <a^0>
n a^0 is the subarray of a (*not* a copy of the subarray) starting at position i and of length j. If i+ j > length (a) this causes a range error.

`.aput` . . . . <a><i><v> / *see comment*
n Store v in the i-th position of a, if 0 i< length (a).

`.aget` . . . . <a><i> / <v>
n Get v from the i-th position of a, if 0 i< length (a).

`.aload` . . . . <a> / <x$_0$><x$_2$>...<x$_{i\ 1}$ ><a>
n (where a is of length i)

`.astore` . . . . <x$_0$><x$_2$>...<x$_{i\ 1}$ ><a> / <a with x$_0$,x$_1$,...,x$_{i\ 1}$ stored in it> n (a is of length i)

`.arrayforall` . . . . <a><x> / *see comment*
n Puts a[i] on the stack and executes x for each object a[i] in a.

`.length` . . . . <a> / <i> n Replaces array a with its length.

`.acopy` . . . . <a> / <a><a> n Duplicates the array a on the stack.

`.dictstck` . . . . / <a> n The array a contains the dictionary stack.

`.execstck` . . . . / <a> n The array `a` contains the execution stack.

`.makeob` . . . . / n All following `JaM` commands and their arguments are saved into an array, until `.stopob` is encountered.

`.stopob` . . . . / <a> n Terminates the above and pushes the resulting array `a` on the operand stack

`.drawob` . . . . / <a> n Executes the `JaM` commands saved in the array `a` by `.makeob` and `.stopob`.

12. Input/Output Commands

This category of commands deals primarily with string and stream objects. Files are represented in `JaM` as special objects called streams. There are three kinds of streams: byte streams for ordinary les of characters, word streams for les of `16` bit words, and keystreams for input from the keyboard. Commands for reading and writing les accept string type arguments and return string results.

There is always an input stream and an output stream. By default, these are both identi ed with the terminal. When writing strings to a le (stream), keep in mind that carriage returns can be part of strings and no implicit carriage returns are ever written to an output stream. Either the input stream or the output stream can be directed to any le. It is also possible to convert streams to strings and manipulate them that way. (see Type Conversion)

`.print` . . . . <s> / *see comment*
n Print the string `s` on the current output stream

`.version` . . . . / *see comment*
n Print a message indicating what version of `JaM` this is.

`=` . . . . <x> / *see comment*
n Utility function from the le `util.jam` which converts `x` to a string and prints it, followed by a carriage return. (Uses `.cvis` see **Type Conversion** .)

`/stk` . . . . / *see comment* n Print the the contents of the operand stack without destroying it. This uses =’’ to print the entries.

`==` . . . . <x> / *see comment* n Utility function from the le `util.jam`. It functions like `=` for printable strings, but it prints useful information about non-printable objects and even uses a reverse dictionary to decipher commands. This dictionary is de ned with the `/buildcommands` function in `util.jam` when `JaM` is initialized, so subsequent de nitions will not be included in it.

`/pstk` . . . . / *see comment*
n Prints the operand stack like `/stk`, except it uses `==`. This may not be good for debugging since `==` does a dictionary look up which could cause an error.

`?` . . . . <s> / *see comment* n Looks up the string `s` in the `$help` dictionary and prints the informative message it nds there. The string `s` should be one of the system commands or utility functions or something de ned using the `/def` utility (see **Dictionary Related Commands**).

`??` . . . . / *see comment*
`n` Print all the keys in the `$help` dictionary (the commands for which help is available).

`.bytestream` . . . . < le name><access> / <bs> `n` This command creates a byte stream with the access characteristics represented by <access> = `1` for read, `2` for write, `4` for append or the sum of any of these. Byte streams are the proper type of streams for most text les. The created stream is left on the operand stack. If access is greater than one, it becomes the current output stream.

`.wordstream` . . . . < le name><access> / <ws> `n` This is like `.bytestream` except the items in the stream are words instead of bytes. This is not appropriate for text les or string I/O.

`.mystream` . . . . / <ks> `n` This command searches the execution stack for the rst stream object and pushes it on the operand stack. Normally this will be the keyboard stream. When you read something from this stream, `JaM` sits and waits for you to type something.

`/altfile` . . . . / *see comment* `n` The user is asked for a le name and subsequent print commands apply to that le. This is a utility function based on `.bytestream`.

`.killstream` . . . . <t> / *see comment* `n` Kill the stream `t` For output streams this restores the destination of `.print` to the terminal.

`/endalt` . . . . / *see comment*
`n` Uses `.killstream` to stop printing to alternate le.

`.run` . . . . < le name> / *see comment*
`n` The le is read as a byte stream and executed. This is how to read text les of `JaM` function de nitions. A *’’ prompt is printed each time the scanner comes to a carriage return.

`.loadbcd` . . . . < le name> / *see comment*
`n` Load Mesa bcd’’ (binary le) and start. This is like `.run` except it runs a Mesa program.

`.debugbcd` . . . . < le name> / *see comment* `n` Like `.loadbcd`, but calls the Mesa debugger before starting the bcd.

`.readline` . . . . <t> / *see comment* `n` <line from stream><`.true`> if the stream `t` is not empty, <`.false`> otherwise.

`.readitem` . . . . <t> / *see comment*
`n` <item from stream><`.true`> if the stream `t` is not empty, <`.false`> otherwise. (An item is one byte for a bytestream or keystream and one word for a wordstream. Bytes are returned

as integers, e.g. ascii codes for characters.)

`.writeitem` . . . . <t><s> / *see comment*
n The item s (see above) is appended to the stream t

`.writebytes` . . . . <t><s> / *see comment*
n Write bytes in string s to stream t

13. Type Conversion Commands

There are several di erent types of objects in `JaM`. String objects have a xed length once they are created. It is possible to change existing strings (see **Scanner and String Manipulation Commands**) but their length remains constant. There are three numeric types: integer, long integer, and real. When the scanner nds a string without any decimal point, it tries to make it an integer, then if it's too big, a long integer. Strings too long to be long integers are converted to reals. Strings containing decimal points naturally become real type objects. It is not possible to enter numbers in exponential notation; however, one can type `6.7 10 -11 .exp .mul` to get 6.7 10^{11} .

Type conversion commands allow the user to determine the types of objects and convert from one object type to another. Type mismatches cause run-time errors. These errors cause special error routines to be executed, which are user de nable and originally come from a le called `errordefs.jam` which `start.jam` reads when `JaM` is initialized.

`.type` . . . . <x> / <Name of Type>
n Deliver the type of <x> on top of the operand stack. Current types include: `.nulltype`, `.integertype` (16 bits), `.longintegertype` (32 bits), `.realtype` (32 bits), `.booleantype`, `.stringtype` (up to 2^{15} 1 characters), `.streamtype` (les), `.arraytype` (one dimensional), `.dicttype`, `.commandtype`, `.stacktype`, `.frametype`, and `.marktype` (stack marker).

`.itype` . . . . <x> / <typenumber>
n Deliver the number of the type of the object on the top of the operand stack. Current assignments are:

`nulltype = 0`	`integertype = 1`
`longintegertype = 2`	`realtype = 3`
`booleantype = 4`	`stringtype = 5`
`streamtype = 6`	`commandtype = 7`
`dicttype = 8`	`arraytype = 9`
`stacktype = 10`	`frametype = 11`
`marktype = 12`	

`.length` . . . . <x> / <i> n Length i of: string (in characters), array (in elements), dictionary (in entries), else 1.

`.cvs` . . . . <x> / <s> n Convert to string equivalent. Applies to numbers, strings, executable strings, and streams.

`.cvis` . . . . <x><s> / <s>
n This is like `.cvs`, except for numbers it destroys the string s,

reusing the space for the result. If the string `s` is too short, `JaM` calls the function `.sizechk` to handle the error. You must define this yourself.

`.cvrs` . . . . <n><r> / <s> n This is exactly like `.cvs`, except if `n` is a number it comes out in base `r`.

`.cvos` . . . . <n> / <s> n The number `n` is converted to a string giving its octal representation.

`.cvirs` . . . . <n><r><s> / <s>
n This is like `.cvis` except numbers come out in base `r`.

`.cvi` . . . . <x> / <i>
n (converts numbers in any form to type integer)

`.cvli` . . . . <x> / <i>
n (converts numbers in any form to long integer type (two words))

`.cvr` . . . . <i> / <x>
n (converts numbers in any form to real type)

`.cvx` . . . . <s> / <s′>
n Converts strings or arrays to executable form. They become verbs. This is the way to get an executable string or array on the operand stack as is required by the execution control commands.

`.cvlit` . . . . <x> / <l> n This converts strings and arrays into literal form (makes them nouns). This undoes the effect of `.cvx`.

`.litchk` . . . . <x> / *see comment* n <`.true`> if `x` is a noun (i.e. not executable), otherwise <`.false`>.

The most important type conversion is the conversion to executable. This is necessary every time a function is defined. There are two main executable forms in `JaM`: strings and arrays. Strings are easier to use, but arrays are faster. When a string is expected, the scanner must extract the objects from the string and each identifier must be looked up in the dictionary stack. When arrays are executed, they are already sequences of objects.

Executable strings are simpler to use because it is easier to change the definitions of the functions they call and they are easier to read and print. The use of the functions `.cvx`, `.load`, and `.exec` when defining executable arrays can be very confusing. Recursive functions are also more difficult with arrays. The following examples illustrate the difference in format:

```
(average) (.add 2 .div) .cvx .def
(average) [ (.add) .load 2 (.div) .load ] .cvx .def
```

When using executable arrays, it is necessary to `.load` each function to get its definition (see **Dictionary Related Commands**). There is a utility called `/compile` to make this a little easier. Use `?` (explained under **Input/Output Commands**) to find out about this.

14. Scanner and String Manipulation Commands

In addition to searching and manipulating strings, the commands listed here allow access to the `JaM` scanner. The scanner rst parses the input into tokens. Tokens consist of blocks of characters separated by tabs, spaces, carriage returns or commas, or anything in balanced parentheses (in which case no separators are necessary). Strings are returned in nal form without the surrounding parentheses. All other tokens are exactly as they appear in the input stream.

`.token` . . . . <s> / *see comment*
n <r><t><.true> if a token is present in the string s, <.false> otherwise. (<r> is the remainder of string s, and <t> is the rst token in s.)

`.string` . . . . <i> / <s>
n <s> is a new string of length i Until something is put in s, it prints as a series of blanks.

`.length` . . . . <s> / <i>
n Replaces the string s with its length in characters.

`.substring` . . . . <s><i><j> / <s^{0}>
n <s^{0}> is the j character substring of s (*not* a copy) starting at positon i We must have 0 i< length (s).

`.putstring` . . . . <t><i><s> / <t^{0}> n <t^{0}> is the same as <t>, except for the substring s starting at position i The rst position is numbered zero and s is not allowed to extend beyond the bounds of t Note that the original t is destroyed.

`.search` . . . . <t><s> / *see comment* n <a><s><b><.true> if there is a substring of t matching s, with b and a respectively the parts of t before and after the rst occurrence of s, and <t><.false> if no substring of t matches s.

`.asearch` . . . . <t><s> / *see comment*
n <a><s><.true> if a starting substring of t matches s (a is the rest of t), <t><.false> otherwise.

15. Graphics Commands

The graphics commands are not part of the basic `JaM` system. The basic graphics primitives are enabled by loading `JamGraphics.bcd`. There are useful extensions to these basic graphics commands in various les of utilities. The basic primitives are implemented in Mesa and provide for an entity called the *display context*, which holds state information concerning the graphics device. Most transformation, clipping, and painting commands alter this state. Drawing commands both use this state and alter it. In addition, the graphics package provides for a display context stack. This is maintained so that transformations and other state information can easily be saved and restored. The `.pushdc`, `.popdc` and `.initdc` commands control this stack.

One of the more important components of the state is the current de nition of the coordinate system. Points in the graphics display are referred to by pairs of real numbers. `JaM` maintains a transformation matrix for converting these numbers to the coordinates used by the display device.

This matrix implements a general a ne transform which can be used to give any combination of tranlations, rotations, and scalings. Changing this matrix will a ect the placement of new objects on the display.

The state information also contains the position of a special point called the *current draw position*. The line drawing commands use the draw position to de ne one endpoint of the line. The draw position is also used to control the placement of text within the display.

The le `Graphics.jam` contains de nitions useful to the beginning user. There are also other de nitions which are intended more for demonstration purposes. A sampling of functions from `Graphics.jam` will be included in the following table along with the basic commands from `JamGraphics.bcd` in terms of which they are de ned. The commands whose names start with a period come from `JamGraphics.bcd` and the other commands come from `Graphics.jam`. It is possible to look at the de nitions of the commands from `Graphics.jam` using `.load ==` (see **Dictionary Related Commands** and **Input/Output Commands**). This allows you to see exactly what these commands do and examine how the more basic graphics commands are used.

There are several concepts common to many graphics commands. First of all, parametric cubic splines are used to specify curves. Cubic splines in turn are speci ed either in terms of the x and y coordinates of points, or by the coe cients of the actual parametric equations. Usually, the coordinates come from the mouse. A cubic spline segment may be speci ed by four points called its Bezier points. They have a speci c mathematical relationship to the spline; informally, the spline passes through the rst and fourth point, goes near the other two, and is always contained within the convex quadrilateral de ned by the four points.

Many commands facilitate the building of paths made up of curves and straight lines. The path does not actually show up on the screen until a command is given to ll in the area enclosed by the path. There are commands which control how the area is lled in. In particular, some commands set up a clipping box which is intersected with the region to be lled in. There is also a painting function which controls the texture (halftone, etc.) of the shaded region.

`.initdc` . . . . / *see comment*
n Initializes the stack of display contents.

`.pushdc` . . . . / *see comment* n Pushes a copy of the current display context onto the display contents stack.

`.popdc` . . . . / *see comment*
n Replaces the current display context with that on top of the display context stack.

`.translate` . . . . $\langle x\rangle\langle y\rangle$ / *see comment*
n The origin of the new coordinate system is set to the location of (x,y) in the old system.

`.scale` . . . . $\langle s_x\rangle\langle s_y\rangle$ / *see comment*
n The coordinate system is expanded by s_x in the x direction and s_y in the y direction.

`.rotate` . . . . $\langle\ \rangle$ / *see comment*
n The coordinate system is rotated clockwise by degrees.

`.sixpoint` . . . . $\langle x_0\rangle\langle y_0\rangle\langle x_1\rangle\langle y_1\rangle\langle x_2\rangle\langle y_2\rangle\langle x_3\rangle\langle y_3\rangle\langle x_4\rangle\langle y_4\rangle\langle x_5\rangle\langle y_5\rangle$ /
see comment n Transform the coordinate system so that the

rst three points are mapped into the the last three. (A general a ne map.)

`.concat` . . . . / *see comment* n Postmultiply the current transformation matrix by another transformation matrix.

`.drawto` . . . . <x><y> / *see comment* n Draw a line from the current draw position to the point (x,y). This moves the draw position also.

`.rdrawto` . . . . <x><y> / *see comment*
n A line is drawn from the current draw position to the sum of that position and the vector (x,y). This moves the draw position as well.

`.linewidth` . . . . <x> / *see comment*
n Sets the current line thickness to x.

`.moveto` . . . . <x><y> / *see comment*
n The current draw position becomes the point (x,y).

`.getpos` . . . . / <x><y> n Put the current draw position on the stack.

`.rmoveto` . . . . <x><y> / *see comment*
n The vector (x,y) is added to the current draw position.

`.drawboxarea` . . . . . . $<ll_x><ll_y><ur_x><ur_y>$ / *see comment*
n A lled box is drawn with the given lower left corner and upper right corner. The draw position is left at the lower left corner.

`.drawcubic` . . . . $<C_0^{(x)}><C_0^{(y)}><C_1^{(x)}><C_1^{(y)}><C_2^{(x)}><C_2^{(y)}><C_3^{(x)}><C_3^{(y)}>$ / *see comment*
n A cubic is drawn with the given coe cients in its parametric equation.

`.beziertocubic` . . . . . $<x_0><y_0><x_1><y_1><x_2><y_2><x_3><y_3>$ / $<C_0^{(x)}><C_0^{(y)}><C_1^{(x)}><C_1^{(y)}><C_2^{(x)}><C_2^{(y)}><C_3^{(x)}><C_3^{(y)}>$
n Four Bezier control points are converted to the parametric form of a cubic.

`.cubictobezier` . . . . . $<C_0^{(x)}><C_0^{(y)}><C_1^{(x)}><C_1^{(y)}><C_2^{(x)}><C_2^{(y)}><C_3^{(x)}><C_3^{(y)}>$ / $<x_0><y_0><x_1><y_1><x_2><y_2><x_3><y_3>$
n The parametric form of a cubic is converted to its four Bezier control points.

`.startpath` . . . . / *see comment* n A path is started for the area generation machinery. Subsequent `.enter`, etc. commands will add to the path. The interior of the path is determined by a winding number technique.

`.starteopath` . . . . . . / *see comment*
n A path is started for the area generation machinery except that even/odd parity is used to determine interior. There is a di erence only in paths which cross themselves.

`.enterpoint` . . . . <x><y> / *see comment* n This command, which must be preceded by `.startpath`, enters the point (x,y) in the current path. Successive `.enterpoint`'s create polygonal paths.

`path` . . . . <i> / *see comment*
n Enter a path de ned by i calls to the `.touch` function. This is equivalent to i repetitions of: `.touch .enterpoint`''.

`.enterrect` . . . . <ll_x><ll_y><ur_x><ur_y> / *see comment*
n The rectangle de ned by the given lower left and upper right corners is entered into the current path.

`rect` . . . . / *see comment*
n The rectangle de ned by two calls to the `.touch` function (see below) is entered into the current path.

`.entercubic` . . . . <$C_0^{(x)}$><$C_0^{(y)}$><$C_1^{(x)}$><$C_1^{(y)}$><$C_2^{(x)}$><$C_2^{(y)}$><$C_3^{(x)}$><$C_3^{(y)}$> / *see comment*
n The given cubic (represented in parametric form) is entered in the current path.

`.enterspline` . . . . . . <x_0><y_0><x_1><y_1>...<$x_{n\ 1}$><$y_{n\ 1}$><n> / *see comment*
n Creates an *open* curve through the n given points and enters it in the current path. This curve is a natural spline consisting of n 1 cubics.

`.entercspline` . . . . . . <x_0><y_0><x_1><y_1>...<$x_{n\ 1}$><$y_{n\ 1}$><n> / *see comment*
n This is the same as enterspline, except a *closed* curve is formed, composed of n cubics.

`spline` . . . . <i> / *see comment*
n This command from the le `graphics.jam` is equivalent to `.enterspline` except the i points come from i calls to the `.touch` function (see below).

`cspline` . . . . <i> / *see comment*
n This command from the le `graphics.jam` is equivalent to `spline` except a closed curve is formed.

`.newboundary` . . . . . . / *see comment*
n A new boundary is started on the current path. If the new part of the path is inside a previous boundary in the same path, it designates a hole in the center when it winds in the opposite direction.

`.drawarea` . . . . / *see comment* n Fill the interior of the boundaries comprising the current path. The current path is then deleted.

`.drawscreenarea` . . . . . / *see comment* n Fill in the whole (but clipped) screen using the current texture and painting function.

`.erase` . . . . .erase / *see comment*
n Erase all area inside the current clipping regions.

`.cliparea` . . . . / *see comment* n Set the clipping region to the interiors of the boundaries comprising the current path.

`.clippedcliparea` . . . . / *see comment* n Set the clipping region equal to its intersection with the interiors of the boundaries comprising the current path.

`clip` . . . . <i> / *see comment* n Start a new polygonal path de ned by `i` calls to the `.touch` function (described below) and make this the current clipping region (from `graphics.jam`).

`cclip` . . . . <i> / *see comment* n Execute the `clip` command except use the intersection of the new region with the old clipping region (from `graphics.jam`).

`clips` . . . . <i> / *see comment*
n The new clipping region is the union of `i` rectangles each de ned by two calls to `.touch` (see below). Each rectangle is de ned by two `.touch`'s: one at its lower left corner and the other at its upper right.

`blob` . . . . <i> / *see comment* n Start a new path and enter a closed spline de ned by `i.touch`'s (like `cspline`) and ll it in using `.drawarea` (from `graphics.jam`).

`cblob` . . . . <i> / *see comment*
n Draw a blob de ned by `i .touch`'s as the `blob` command does except cause it to be intersected with the current clipping region (from `graphics.jam`).

`poly` . . . . <i> / *see comment* n Draw a polygonal line de ned by `i` calls to the `.touch` function (from `graphics.jam`).

`.dot` . . . . <x><y> / *see comment*
n Put down a dot at the coordinates speci ed.

`dot` . . . . <x><y> / <x><y>
n This command form `graphics.jam` is like `.dot` except it copies its arguments.

`area` . . . . <i> / *see comment*
n Fills in polygonal area de ned by `i` calls to `.touch`, putting a dot at each `.touch` (from `graphics.jam`).

`.texture` . . . . <n> / *see comment*
n The integer `n` is viewed as a sixteen bit octal number. It de nes a `4` `4` bit pattern which is tessellated when lling in areas. By default, `n =` `1` is used. This causes areas to be lled in solid black.

`.paint` . . . . <n> / *see comment* n This function interacts with the `.texture` setting to determine how pixels are reset when lling in areas. The integer `n` is in the range `0...3`, but is more convenient to use the identi ers de ned in the le `graphics.jam`. Pixels for which the texture bit is `1` are: set to 1 (black) if `n = 0` (paint), inverted if `n = 1` (invert), set to 1 and all other pixels are set to 0 if `n = 2` (replace), and set to 0 (white) if `n = 3` (erase). The default is paint.

`.touch` . . . . / <x><y>
n Returns the current corrdinates of the mouse when the red'' button is pushed. Execution is halted while waiting for the user to push the button.

`.mouse` . . . . / <x><y>
n Returns the current coordinates of the mouse in the current coordinate system. Execution does not halt. No buttons need be pushed.

`.redup` . . . . / *see comment* n This is a user de nable function which is executed every time the red'' mouse button is released and the keystream is empty. When this happens, the coordinates of the mouse are put on top of the stack. Then the function is executed. All of these default to `.pop .pop` so they originally function as no-ops.

`.reddown` . . . . / n *similar comment*

`.yellowup` . . . . / n *similar comment*

`.yellowdown` . . . . / n *similar comment*

`.blueup` . . . . / n *similar comment*

`.bluedown` . . . . / n *similar comment*

`.setfont` . . . . <KS-font le name><n> / *see comment* n Set the font for the character drawing commands, using n point type. There is no default font, so this command is required before any characters can be drawn.

`.drawchar` . . . . <c> / *see comment*
n Draw a character at the current draw position and update the draw position to re ect the width of the character.

`.erasechar` . . . . / *see comment* n Erase a character at the current draw position and update the draw position by the negative of the width of the character. (Not currently implemented.)

`.drawstring` . . . . <s> / *see comment*
n Draw the string s at the current draw position and update the draw position to re ect the length of the string.

`.getcharbox` . . . . <c> / <sizex><sizey><orgx><orgy><widx><widy>
n Return the bounding box and width of the given character.

`.getstringbox` . . . . . . <s> / <sizex><sizey><orgx><orgy><widx><widy>
n Return the bounding box and width of the given string.

`text` . . . . <x><y><d_y><s> / *see comment*
n Put the lines of text from the string s into the screen starting

at the point `(x,y)`. Lines are separated by carriage returns and are spaced d_y below each other with a left margin at `x` (from `graphics.jam`).

`.displayoff` . . . . / *see comment*
n Blank out the whole screen including the text window.

`.displayon` . . . . / *see comment*
n Reverse the e ect of the previous command.

`.snap` . . . . < le name> / *see comment* n Creates a press le of the speci ed name containing an Alto-resolution image of the graphics portion of the Alto screen.

`.openpress` . . . . / *see comment*
n Open a press format le. Graphics commands will now cause the appropriate instructions to be placed in the press le (i.e., things will not be drawn on the screen). This command and `.closepress` start a new display context.

`.closepress` . . . . / *see comment* n Close a press format le.

16. Error Handling

Run time error handling routines in `JaM` are user de nable. When such an error is detected, the appropriate function (identi er, whose value is to be executed) is called. The default de nitions of these functions are found in the le `errordefs.jam`. Not having these defaults loaded will cause an in nite string of errors when `JaM` tries to call the error handling routines. These defaults may be changed, but typically they stop execution and print the stack after displaying a short message regarding the type of error.

When writing new error handling routines one must keep in mind that the operand stack and execution stack must remain intact when the error handling routine is called. One version of these routines prints the stack using `/stk` (see **Input/Output Commands**) and puts the user in a loop which reads lines from `.mystream` (the input stream) and executes them. This terminates when an empty line is found. Hitting carriage return causes `JaM` to attempt to continue execution. At various stages in this routine it is possible to get an additional (nested) error. This could be a little awkward. In this case it is a good idea to use the `.stop` command. The error handling routines are:

`.stkundflow` . . . . / *see comment* n Called when there is an attempt to get something from an empty stack.

`.undefkey` . . . . / *see comment*
n Called when an identi er cannot be found in a dictionary.

`.longname` . . . . / *see comment*
n A rare error caused by excessively huge le names.

`.badname` . . . . / *see comment*
n Called when a string which was supposed to be a le name cannot be found in the directory.

`.typecheck` . . . . / *see comment* n Called when some argument of a function is of the wrong type. The o ending primitive command is left on the stack along with its other arguments.

`.dictfull` . . . . / *see comment*
n Called when an attempt is made to de ne something into a full dictionary. This may mean a system utility has tried to de ne its own temporary variable into a full user dictionary.

`.syntaxerr` . . . . / *see comment*
n Called when the scanner nds an unmatched)''.

`.overflow` . . . . / *see comment* n Arithmetic over ow.

`.stkovrflow` . . . . / *see comment*
n The total number of entries on the operand, execution, and dictionary stacks cannot exceed `256`

`.rangechk` . . . . / *see comment* n Something out of range, usually in an array or string manipulation command.

`.sizechk` . . . . / *see comment*
n Occurs only in the `.cvis` and `.cvirs` commands (see **Type Conversion**). Warning this routine has been neglected in some versions of the le `errordefs.jam`.

17. The Edit Package

This package can help reduce some of the inconvenience of continually having to change a bravo le when one is debugging programs in `JaM`. It is possible to make minor changes in function de nitions without having to exit `JaM` and `.run` an external le again. This helps prevent the virtual memory from being lled up with garbage and it can save a lot of time. The editor is necessarily rather primitive, however, and it is desirable for safety to have programs saved on external les. For these reasons, this package should not be overused.

To use this package, run `edit.jam` (see `.run` under **Input/Output Commands**). The commands are:

`/edit` . . . . <k> / *see comment* n Tell the editor to edit the value of the key `k` (usually a function name). <k> and its value are printed in the format of `/p` (see below). Subsequent calls to `/p` and `/r` refer to `k`.

`/p` . . . . / *see comment*
n Print the function de nition being edited. The format is: (name)(de nition).

`/r` . . . . <s><r> / *see comment*
n The rst occurance of the string `s` in the de nition being edited is replaced by the string `r`. If `s` is not found, a message is printed and the de nition is unchanged. To see the e ect of your change, use `/p`.

18. Programming and Use of JaM

Because `JaM` is quite di erent from other programming languages, it is appropriate to give hints on how to program and put `JaM` to good use.

The `JaM` input language has very little syntax. This attribute is both good and bad. The lack of syntax is good because a uniform representation is attained. The lack of syntax is bad because the code is hopelessly unreadable. In `JaM` it is almost true that any line of code can do anything (given the appropriate rede nitions of identi ers). Because of this property of `JaM` code, it is desirable to do several things when programming. First, document each function as it is written (use `/def` and `/xdef` under **Dictionary Related Commands**). Second, use naming conventions for functions that all belong to a given class (for example, intrinsic commands are started with `.`’’). The latter convention will allow a person reading the code to tell what category a function is in by its name and may help him avoid accidentally rede ning intrinsic functions.

Since `JaM` is a stack oriented machine, the user must mentally keep track of the contents of the stack. It is easier to program `JaM` if each routine performs only one function and is very short. Normally a `JaM` routine should be at most one or two lines long. For example, suppose we wish to use the absolute value of a given number. Then, rather than introduce the code in line, it is desirable to write an "abs" function and then use that function e.g.

```
(abs)(the absolute value of a number)(.dup 0 .lt (.neg) .cvx .if) /xdef
```

Also if complex parameters are being passed to `JaM` control statements, it is better to name the parameters rather than have lines and lines full of parentheses.

B