XEROX

BUSINESS SYSTEMS

Systems Development Division

November 26, 1977

To: User Interface Group

From: Dave Smith / SD at Palo Alto

Subject: On Janus Procedures

Filed on: <davesmith>janus-procedures.press

Disclaimer: This paper has no relationship to the section that will go in the Janus Functional Spec. It is strictly a "think piece" discussing what I think are interesting ideas concerning procedures. The Functional Spec section will, of course, omit all of the philosophical discourse (otherwise known as bullshit).

Second Disclaimer: There are various ideas floating around the Janus program as to what "procedures" should be used for:

- automating repetitive tasks (me, Paul)
- data management (Richard, Peter Bishop, Linda, Ron Johnson?)
- information flow control (OfficeTalk, Richard?, Peter Bishop?, Ron Johnson?)
- others?

I don't think these are necessarily disjoint; in fact I expect the procedure design discussed here to be applicable to all of the application areas above, certainly by the second or third release of Janus. However focusing on an application has tended to lead the designers down different paths. I am concentrating on procedures as *recorded sequences of user actions*, i.e. as macros. Therefore, to clarify that this is what I mean by "procedure", I will use the term "user macro" or simply "macro" instead of "procedure" in this memo. Hopefully this will integrate with whatever other procedure facilities people implement (e.g. records processing).

Goals

The user macro design described here is intended to accomplish two goals:

1. To permit the *average user* to "program" his Janus system. The main purpose of the programming is to *automate his repetitive tasks.* A user who cannot program his computer can only exploit a fraction of its power.

2. To make the programming *no more difficult than the normal operation of the system.* This goal is, of course, not absolutely achievable; however I believe that this design comes close.

If we can achieve goal #1, then programming will be available -- for the first time -- to a vastly larger class of users. If we can achieve goal #2, then our users may find themselves doing quite a lot of programming. Ideally they will come to rely heavily on programs in their everyday work habits. Only when this happens can we rightly say that we have "automated the office".

Background

There are in existence today several good models for the type of user programming which I would like to see incorporated into Janus (at least in the first release -- later releases will probably require more complicated concepts).

The first thing to observe is that goal #2 has already been attained in limited contexts. Consider Bravo.

Bravo "replay" transcripts

Bravo builds up a "program" in the background during every editing session. A replay transcript is a "program" under any reasonable definition because it uses one or more inputs (text files), contains a machine-executable sequence of instructions (the Bravo commands), and produces one or more outputs (text files). (The fact that it operates only on text files and only on the *specific* text files for which it was defined does not alter the fact that a replay transcript is a program; it is simply a limited-capability program.)

The fact that Bravo builds a replay file in the background *does not make Bravo any harder to use* than if it didn't build the file. Secretaries could use Bravo in either case. (But a question to be investigated is whether *generalizing* the replay transcript notion will keep it as easy to use.)

The next thing to observe is that goal #1 also has already been attained by various systems. One of the earliest was the Unimate "robot". The following description of the Unimate is taken from my thesis. It illustrates that a person does not have to be a "computer scientist" to program a computer; in the Unimate case the programmer is a machinist.

Unimate robot

"The Unimate robot consists of a mechanical arm with 6 degrees of freedom mounted above a large base containing electronics. It is a programmable manipulator designed for industrial applications.

"The robot may be operated in either of two modes: training mode or production mode. In training mode, the robot's are is *guided through the steps necessary to perform a task* by a human 'trainer'. The robot has a digital electronic memory in which it can 'remember' up to 1024 operations and their timing. Typical operations are 'rotate a joint',

'move to (x,y)', 'close the hand', etc. After the training phase the robot can operate in production mode, automatically repeating the operations in its memory. It will repeat the operations indefinitely, until stopped or until a pathological condition occurs.

"The robot has been particularly successful on assembly lines. General Motors is presently using 26 Unimate robots to do 80% of the final welding on its Vega assembly line. The primary practical deficiencies of the robot are the absence of (a) conditional branching and (b) force or visual feedback. From a conceptual standpoint, the robot is a relatively uninteresting computer since its only data structures are (x,y) coordinates.

"However the Unimate demonstrates the potential of dynamic, analogical programming. Rather than writing an algorithm whose form bears no resemblance to the task to be done, programming occurs by actually *doing* the task. Advantages:

"(1) The *act* of programming is analogous to the *function* of the program being written. It is *learning by doing.* Programming is exceptionally clear and easy.

"(2) An untrained operator can program the robot, 'untrained' in the sense that he need have little knowledge of computer programming -- he need only be familiar with the task the robot is to perform. This makes the robot accessible to a large class of users.

"(3) Bug-free programs can be written the first time. Since programming involves doing the task once, *successful completion* of the task means that a correct program has been written (modulo mechanical and/or timing inaccuracies)."

These three characteristics are also desireable for programming in Janus. The principal advance of the Unimate is its "analogical programming," in which the *act of programming* is analogous to the *purpose of the program.* Another example of anological programming is the Hewlett-Packard HP-65 pocket calculator. (The following description is paraphrased from my thesis.)

HP-65 Pocket Calculator

This was the first (and in my opinion is still the best) of the hand-held programmable calculators. Programs are written simply by putting the calculator in "program" mode and then pushing the desired sequence of keys, *just as if one were doing the calculation.* The key-pushes are remembered on a magnetic strip. The magnetic strips can be taken out, stored "off-line", and read back in later. Hewlett-Packard has published a library of standard programs, and users typically make their own additions. In "run" mode the programs can be executed or the calculator can be operated manually.

However a major defect in the design is that in "program" mode the display does not show the current state of the calculation. Rather it shows a numerical representation for the last key pushed (e.g. the 17th key). So programs normally have to be designed in "run" mode, written down, and then entered in "program" mode. One cannot construct anything but simple programs directly in "program" mode because it is too abstract.

Officetalk "edit sequences"

Officetalk has also developed analogical programming in the context of forms fill-in. In the ORG memo "A Forms Editor for Officetalk" (Oct. '76) Newman and Mott identified three constraints which they felt a forms editor must satisfy:

> "1. Officetalk distinguishes between those areas of a document that may be altered by the user, and those that may not. The editor must take this distinction into account.
>
> "2. The user needs some way of repeating the sequence of edits involved in forms entry and editing, in order to simplify the task of filling in the same form next time.
>
> "3. The user needs a 'calculator' for forms editing."

The last two are particularly relevant to Janus macros. Newman and Mott described a scheme for capturing user actions in an "edit sequence":

> "As the user edits, an edit sequence is built up in a special window. Each edit is in the form of an 'assignment statement' indicating which field is filled in, and which fields supply the information; edits that involve only parts of fields are not added to the edit sequence. The next time the user performs the same task on the same form, the previous edit sequence is retrieved. He can then either perform a different sequence of edits, or use the old sequence. To use the old sequence, he can either pick each edit from the displayed sequence, or use the *next-edit* command (ESC key) to step through the sequence."

A typical edit sequence is:

```
Total ← Total + amount
Due ← Advance - Total
Signature ← input
Date ← OZ.date
Branch ← '580'
```

This is constructed by *remembering* the actions as the user does them and by *generalizing* from the specific field used to the generic field it represents. Newman and Mott developed a clever way to enter constants and run-time-needed inputs into edit sequences (cf. the above mentioned memo). Janus user macros will incorporate something like this. Note that their edit sequences record only actions that modify or use an entire field's contents; intra-field edits are ignored. This is in sharp contrast to Bravo, which records everything. My inclination is towards the Bravo interpretation.

Pygmalion "dynamic programming"

Another example of an articulate method of defining algorithms is the work in my thesis (ahem): *PYGMALION, A Computer Program to Model and Stimulate Creative Thought.*

(Incidentally, it is now available from Birkhauser Verlag if you're interested.) In PYGMALION I developed visual, interactive counterparts for the standard programming concepts: conditionals, iteration, subroutines, etc. Since PYGMALION's emphasis was on modelling existing computer science concepts, the work is not directly translatable to Janus. (Janus is not aimed at computer scientists.) However, I think it provides an existence proof that there are solutions to such problems as finding an analogical way to do conditionals.

In PYGMALION, as in the systems above, there is a "remember" mode of operation. In "remember" mode each operation, in addition to executing its internal semantics and updating the display, adds itself to a list of instructions called a "code list". A code list could be associated with an icon and re-executed on demand. Code lists contain only generic operations, as in Officetalk; PYGMALION *generalizes* from the specific icon used to the generic icon that it represented. This generalization is the core of the problem of defining algorithms by dynamic interaction.

NLS command sequences

On a slightly different emphasis, the NLS (oN-Line System) editor at SRI refined the notion of user-editable interpretive commands. NLS commands are heavily syntax driven; a typical command is:

MOVE <entity> (at) <source> (to) <destination>

e.g.

MOVE WORD (at) BUG (to) BUG

where BUG represents a mouse selection. The "M" and the "W" are typed from the keyboard, much as in Bravo, and the system expands them into the complete command words. The command words are echoed in a feedback window at the top of the display screen, again similar to Bravo.

Though we have explicitly rejected this modal, syntax-driven, teletype-oriented approach, it does have an advantage which NLS exploits well. The user can construct (with the editor) a statement consisting of a sequence of command strings like the MOVE command above. NLS provides a command:

PROCESS (commands from statement at) <source>

with which the user can execute a command sequence. Since *each command in the sequence consists of the text that the user constantly sees in the feedback window,* writing them down is easy and natural. Nearly all NLS users write command statements at some time. In fact, people would broadcast to the whole community command statements of which they were particularly proud or which they found particularly useful. Most people had little difficulty in reading these sequences.

(A deficiency on which we were working when I left the NLS project is that command statements have to be constructed statically with the editor, like any other text statement. We were exploring techniques for *recording* commands as the user does them, but we had not yet implemented it.)

A minor breakthrough pioneered by NLS was its way of describing a point in a document. It developed the notion of a "link". A link is an ordered quadruple of the form:

<host, directory, file, statement>

(Formatting information could also be included in a link.) Any of the first three fields could be omitted, in which case the defaults were: current (Arpanet) host, current (Tenex) directory, current (NLS) file. The statement within a file could be described in several ways: (1) as a position in the NLS tree-structured file, (2) as a number unique to the file (there were actually two flavors of this), (3) as a user-specified name string, or (4) as the statement containing a given string (associative retrieval). A typical type-1 number is "4b3", which says "the fourth top level statement in the file, then the second statement under that, then the third statement under that." This is a *dynamic* description; the 4b3'th statement *at execution time* would be designated.

All commands accept links as operands. For example, the <source> and <destination> in the MOVE command above would be links. The system automatically turns a mouse bug into a link internally, so that *links are the NLS representation for selections.* I believe that Janus may be able to use this concept, though I have not worked out the details.

Principles for Janus macros

To sum up: The above systems work, they are articulate, and they can teach us a lot about programming machines. From their lessons I have extracted the following four principles for Janus user macros:

Dynamic programming (remember mode)

Macro definition is based on the concept of a "remember" mode of operation. In this mode all of the actions that one normally does on the system are available, but in addition to *executing* each action, the system also *records* it.

Analogical programming (learning by doing, DWID)

Macro definition proceeds by actually doing a task once. The user "teaches" the machine what to do in a step-by-step demonstration. With this "Do What I Did" approach the problem of *writing a macro* reduces to the problem of *doing the task you want the macro to accomplish.*

Concrete programming (work with actual data)

Macro definition uses actual data. As a user guides the machine through the desired steps, he does so on actual documents, folders, file cabinets, etc. The machine *generalizes*, where appropriate, from *actual* to *generic* data.

System integration (human-readable macro representation)

Macros are displayable, printable, mailable, fileable, etc. They are displayable in an intelligible fashion to the user, and he can edit them with the standard text editor. The

system also provides dynamic ways to edit and debug macros.

Design

Definition: A *Janus user macro* is a document containing a recorded sequence of Janus actions.

Definition: A *Janus action* is a selection or a command. A command may either be from the keyboard or from a menu. The selection is taken to be the selection that exists when a command is invoked. (Adjustments and re-selections are not recorded.)

Janus user macros are defined via the DWID approach. With one or two exceptions, *all Janus actions can be recorded.*

A Janus macro document has all of the document capabilities (editable, printable, mailable, fileable, etc.). In addition it is *executable.* There is a special rendering of a macro document to indicate that it is executable (see Figure 1).

Every recorded action has a text representation; for example a MOVE command is represented by the string "MOVE". In general the representation is the text on the key top or in the menu. The representation for selections is yet to be determined. Each line in a macro document contains a separate recorded action.

Implications of the design

Janus macros have *exactly the same capabilities* as a user doing the actions manually. The system does not achieve any additional capabilities simply because a macro is running. In fact, there is no difference to the system between a running macro and an (extremely fast) user doing the actions.

There is no formal language for user macros (at least in the first release) beyond a rendering of Janus actions. There is no static control language (conditionals, iteration), declarations, syntax, compound expressions, etc.

Implementation

Two additional icons are introduced in order to implement this design:

RECORDER icon

> This contains the functions necessary for the recording of user macros. The user's model is based on a tape recorder metaphor. The functions and concepts involved are similar to those existing on physical tape recorders, particularly cassette recorders.

CALCULATOR icon

> Following Officetalk's lead, we introduce arithmetic into the system via a pocket calculator metaphor. It is obvious that a user should be able to do arithmetic calculations on a personal computer, quite apart from whether or not there are macros. Thus this is

introduced not solely for the sake of macros but also to remedy an omission in Janus.

The RECORDER

This is the core of the macro facility. The Recorder consists of a window and a set of commands (see Figure 1). As in other function windows, the Recorder window can contain any number of documents and/or folders. Like the Printer and Out Basket, the top document or folder is processed first. *All Recorder commands affect only the top entry in the window.*

In the lower half of the Recorder window is the "Current Actions Area" and the "Next Action Arrow". The Current Actions Area always displays the actions in the top macro document in the Recorder window. (It may be empty if we are going to record a new macro.) The Next Action Arrow always points to the next action that the macro will execute. This is the action with which the PLAY, CONTINUE and SINGLE STEP commands begin executing.

** Note: The contents of the Current Actions Area can be scrolled, *but the Next Action Arrow remains fixed.* Thus a side effect of scrolling is to bring a different action under the Next Action Arrow. This gives the system a kind of primitive GO TO ability (perish the thought).

All Recorder commands can themselves be recorded unless explicitly noted otherwise.

PLAY

To execute a user macro:

(a) select a macro document (or folder containing macro documents)
(b) MOVE or COPY it to the Recorder window
(c) execute PLAY.

PLAY begins execution with the action pointed to by the Next Action Arrow.

Unlike the Printer, the user must explicitly invoke PLAY on each document to be processed (because the user might want to RECORD it or SINGLE STEP it instead). Also unlike the Printer, documents and folders are not deleted once they have been processed; they are simply placed back on the Desktop. (The Printer should not delete them either, just between you and me.)

When a folder is processed, the Recorder executes each document or folder in it in sequence without requiring any user intervention. In this way a set of macros can be packaged up and executed as a unit.

** Note: The PLAY command being recordable gives macros a subroutine capability. A macro can select a macro document, place it in the Recorder, and execute PLAY, all as part of its recorded actions. It can even execute PLAY on itself, producing recursion.

STOP

When RECORDing a macro, STOP ends the recording session. The session can be resumed at any time by executing RECORD again. When PLAYing a macro, STOP aborts it. The only

time the user has the option of executing STOP during the running of a macro is when it has executed a PAUSE. In either case, STOP removes the macro document from the Recorder window and places it on the Desktop, just as if it had finished executing.

When not RECORDing or PLAYing a macro, STOP has no effect.

STOP itself cannot be recorded, since it used to take the system out of record mode.

RECORD

To record (define) a user macro:

(a) select a macro document
(b) MOVE or COPY it to the Recorder window
(c) execute RECORD.

All (recordable) actions will now be inserted into the macro sequence in the Current Actions Area in front of the action pointed to by the Next Action Arrow. (If we are defining a new macro, the Next Action Arrow will not point to any action.)

RECORD, like PLAY, affects only the top document in the Recorder window. (It must be a document; it cannot be a folder.) If the macro document is non-empty, the system prints a warning such as "macro <name> is already defined. Do you want to modify it?" and a YES/NO menu is displayed. If NO, the RECORD command is ignored. If YES, the system begins recording, inserting actions before the Next Action Arrow.

** Note: Macros can be edited by SINGLE STEPping or scrolling to a given action, executing RECORD, and adding new actions to the sequence. Actions can be deleted by selecting them and executing DELETE, as with any other text. Actions can also be MOVEd, COPYed or typed in by hand, but this is somewhat riskier since the user can create illegal actions that way. The safest way to add new actions is to enter record mode and let the system add them. The system only records legal actions.

RECORD itself cannot be recorded, as it would probably be confusing.

PAUSE

The PAUSE command temporarily suspends the running of a macro. When a PAUSE command is recorded, the user is asked to supply a text string to use as a prompt. This string is printed in the Message Area every time the PAUSE is subsequently executed. The user then has the option of

- CONTINUEing it,
- STOPping it,
- SINGLE STEPping through it, or
- doing any number of Janus actions and then one of the above.

** Note: This macro design contains no conditionals. I think that conditionals require considerably more understanding of programming than other Janus actions and that we should defer them at least until the second release of Janus, *even though this severely limits the types of programs that can be written with this system.* Eventually I expect

that we will come up with an articulate method for describing decisions, possibly like the Pygmalion IF operator or perhaps like Richard Moore's records processing decision tables. But I think that we should not attempt to incorporate it into the first release. As an alternative, I suggest we adopt the solution used in the HP-65 calculator. They also have a PAUSE operation. When executed, the programs stops and *the user* looks at the state of things and decides what to do next, either

- aborting the program,
- resuming it,
- going to a different program step, or
- doing any number of calculator operations and then one of the above.

This is a substantial simplification because the user must deal only with a *concrete* situation requiring a decision instead of an *abstract* "every time" situation.

CONTINUE

The CONTINUE command resumes a PAUSEd macro with the action pointed to by the Next Action Arrow. It is similar to PLAY, with one exception: if the system was in record mode when the user executed PAUSE, the system is temporarily taken out of record mode so that the user can do some non-recorded actions. In this case CONTINUE will put the system back in record mode, whereas PLAY will not.

SINGLE STEP

This command permits the controlled execution, debugging and editing of macros. As mentioned above, the Next Action Arrow always points to the next action that the macro will perform. SINGLE STEP causes this action to be executed and the arrow advanced. SINGLE STEP can be invoked when a macro has not yet begun running (i.e. instead of invoking PLAY) or when a macro has executed a PAUSE.

REPEAT

The REPEAT command is introduced as an amittedly clumsy attempt to get iteration into the system. Like CONTINUE, REPEAT is similar to the PLAY command, with one execption: it traps failures and resumes running with the next action following it.

> Every action in a macro either succeeds or fails when it is executed. It succeeds if it works correctly; it fails if it is unable to carry out its semantics. For example, when the last field in a form is selected, NEXT will fail because there are no more fields to go to. *The failure of any macro action automatically aborts the macro.*

This characteristic of macro actions together with REPEAT give the system a kind of iteration capability. REPEAT causes the action under the Next Action Arrow to be executed (i.e. it does a GO TO to that action). Typically the user scrolls the Current Actions Area until the desired action is specified. The action must be an action earlier in the sequence, hence the name "repeat". This will cause the action sequence to loop indefinitely, until finally an action fails. When that happens, the REPEAT action regains control and causes the macro to proceed with the following action.

NEW MACRO

This is simply an optimization for creating a new macro document. When it is executed, the system makes a copy of the System Blank Macro Document, puts it in the top of the Recorder window, and enters record mode. It is exactly equivalent to the following sequence of actions:

- select the System Blank Macro Document (wherever it is)
- COPY
- designate the top of the Recorder window
- RECORD .

It is intended to make defining macros sufficiently easy to be widely used.

Comments

Comments may be added manually to a macro document by inserting comment text enclosed in (parentheses).

The CALCULATOR

The Calculator consists of a window and a set of commands (see Figure 2). It not only permits macros to contain arithmetic operations, but it also remedies the lack of arithmetic ability in Janus.

The calculator model and functions should be very carefully designed, something I have not yet done. The model presented here is just to give you the flavor of what I am thinking about. If anyone has a good model for a calculator, I would appreciate hearing about it. (Sample problems: Should there be a stack or not? Should there be registers? Memory locations? What functions? Log functions? Trig functions? Inverse trig functions?)

In the simple-minded model here, there is a field in the Calculator window called "X" and an internal "accumulator". All commands operate on the current selection as an operand. Unary operations (e.g. square root) operate only on the current selection; binary operations (e.g. addition) operate on both the accumulator and the current selection. If there is no selection when a Calculator command is executed, the X field contents are used as the selection. After every operation, the result is stored in both the accumulator and the X field. In addition, *after every operation the contents of the X field contents are selected* so that operations can be chained easily. Numbers can be entered from the keyboard by typing them into the X field. As an (admittedly inconsistent) optimization, if a number is typed into the X field while its contents are selected, the typed number *replaces* the contents. Otherwise a DELETE would have to be done every time you wanted to type in a number. (On the other hand, virtually all calculators work that way.)

The Calculator can operate on extended selections, in which case it iterates performing the operation on each number in the selection. In particular a row or column of a table can be selected and used as the operand for a Calculator function. Entries that are not legal numbers are skipped.

The Calculator is floating point, to some reasonable precision.

Calculator Commands

Calculator commands are postfix, as are all commands in Janus.

clear	X,accumulator ← 0
+	X,accumulator ← accumulator + selection
-	X,accumulator ← accumulator - selection
*	X,accumulator ← accumulator * selection
/	X,accumulator ← accumulator / selection
%	X,accumulator ← accumulator * selection * 0.01

1/s	X,accumulator ← 1 / selection	
s^2	X,accumulator ← selection * selection	
sqrt	X,accumulator ← SquareRoot(selection)	
10^s	X,accumulator ← 10 ** selection	(10 to the selection power)
log	X,accumulator ← Log(selection)	
e^s	X,accumulator ← e ** selection	(e to the selection power)
ln	X,accumulator ← Ln(selection)	
a^s	X,accumulator ← accumulator ** selection	(acc to the selection power)
sin	X,accumulator ← Sine(selection)	
cos	X,accumulator ← Cosine(selection)	
tan	X,accumulator ← Tangent(selection)	

Calculator Scenarios

To do 3 + 2:

CLEAR (automatically selects X field contents)
(a DELETE would have to go here unless typing into X replaces its contents)
type 3
+
(a DELETE would have to go here unless typing into X replaces its contents)
type 2
+

To do [field 1] ← [field 2] + [field 3]:

CLEAR
select [field 2]
+
select [field 3]
+
MOVE (or COPY) (moves the X field contents)
select [field 1]

To add up a column in a table:

CLEAR
select the column
+ (X,accumulator ← accumulator + $column_i$, i=1 to #rows)

Generalization of selections

The most interesting problem concerning a DWID type interface for defining algorithms is when and how to *generalize from a specific object to the generic object that it represents*. All the analogical systems in existence have had to solve this problem. Even Bravo generalizes in its replay transcript from the particular characters selected to the position of those characters in the file (e.g. 43rd through 86th characters). Thus you can play the transcript on a different source file, and it will run (but it almost certainly won't do what you want).

Janus macros implement generalizations in the following ways. I don't claim that they provide intuitive solutions to all of the applications for which we may want to use macros. But I do think they are the right interpretations for such important applications as filling in forms and direct mail.

Function icon selections

All function icons are given a unique number. (Perhaps we can use or assign a Pilot unique number.) When a function icon is selected in remember mode, its unique number is entered into the macro, so that *that specific function icon will always be used*. We will represent an icon selection as:

<icon #>

e.g.

<icon 7>

Thus there is no generalization done on function icons. If I select my file drawer at define time, my file drawer will be selected at run time.

Document/folder selections

All documents and folders, on the other hand, are given a *relative position number* when they are selected in remember mode. The relative position number is determined in the following way. If the document or folder is in a folder or function window, its position relative to the top of the window is recorded (e.g. the 3rd document down). If it is on the Desktop, the Desktop is treated like a *stack*, so that the last document or folder brought to the Desktop is given relative number 1, the next to last number 2, etc. If a document is nested inside several folders, the relative numbers are chained: the 3rd document in the 4th folder in icon 7. We will represent a document or folder selection as:

<icon # or Desktop #, offset 1, offset 2, ...>

e.g.

<icon 7, folder 4, document 3>

Thus documents and folders are generalized. Whatever the third document in the folder is, it will be selected.

Note: There is nothing that says that only macro documents can be placed in the Recorder window. Therefore, as long as the user does not attempt to execute them, a user can employ the Recorder window as a handy temporary storage place

for documents and folders. They will then be in a known place at run-time and can be easily designated.

Text selections

Text selections are also generalized in the same limited way as Bravo, namely as character offsets from the beginning of a document. We expect that text selections will generally not be recorded, except for short-term special-purpose tasks. An example of a text selection is:

<icon 7, folder 4, document 3, characters 89-125>

Thus text selections are generalized but are not very useful.

Field selections (and other frames?)

Every field in a Janus document has a unique sequence number beginning with 1 for the first field in a document and monotonically increasing. When the *entire contents* of a field are selected, a representation using this sequence number is recorded; for example:

<icon 7, folder 4, document 3, field 16>

This is similar to Officetalk's solution for fields. Thus fields are not generalized within a document. This means that if I rearrange fields or add or delete text around them, a macro to fill in the fields will still work.

If other frames were given unique sequence numbers as well, the above solution would apply to them as well. Thus we should consider changing the Functional Spec. so that sequence numbering applies to *all* frames.

Ways to make selections

Using the MOUSE - The above all deal with generalizing selections *made with the mouse.* There are two additional ways to specify a recordable selection.

Using SEARCH - The SEARCH command finds text strings anywhere in a document (and documents anywhere in a folder?). Thus this command is inherently different than bugging with the mouse. It is both more general and more specific. One the one hand it is more general because no matter where the text occurs in the document, it will be selected. On the other hand it is more specific because it always deals with the same text (or document), never a different one.

Using PAUSE - The PAUSE command is an excellent way to *let the user specify the selection at run-time.* For example, the following action might be recorded:

PAUSE "Please select the address list to use"

When PAUSE is used, the problem of specifying parameters largely goes away.

Enhancements

It is quite likely that users will want to build up macro libraries. Therefore there will

probably be folders containing only macro documents. We may want to take advantage of this with some special capabilities, either in the first release or later. For example, we might permit abbreviation expansion on the name of a macro document to be a way to invoke a macro; name scoping would be controlled in the same way abbreviation names are controlled: by putting the macro documents and/or folders into the Abbreviation Folder. Or we might implement special lookup and invocation functions on macro folders. These areas are up in the air right now, but only the clearest and most beneficial features will make it into the first release.

Standard programming concepts

The following is a summary of how the system implements various programming concepts. Not all concepts are listed of course, and not even all of the ones listed are implemented by Janus macros.

Variables

Implemented by: generalization of selections.

Conditionals

Implemented by: no conditionals in first release. PAUSE command instead.

Iteration

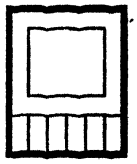
Implemented by: REPEAT command together with action success/failure. (Records processing also incorporates iteration for its uses. When it is more completely specified, we will integrate it with our notion of iteration. Frankly I would like to get rid of the REPEAT command.)

Subroutines, recursion

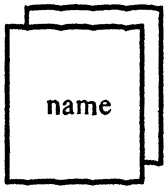
Implemented by: PLAY command.

GO TO

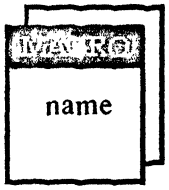
Implemented by: PLAY or CONTINUE together with the Next Action Arrow.



RECORDER icon



Ordinary document



Macro document

RECORDER

	NAME	SIZE	STATUS
	Read Mail Procedure	6 actions	playing
	Answer Mail Procedure	7 actions	
	Fill In Expense Report	29 actions	
	Process Purchase Orders	84 actions	

Current Actions

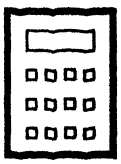
SELECT <icon 7, folder 4, document 3, field 16>
+
SELECT <icon 7, folder 4, document 3, field 17>
+
MOVE

PLAY | STOP | RECORD | PAUSE | CONTINUE | SINGLE STEP | REPEAT | | NEW MACRO

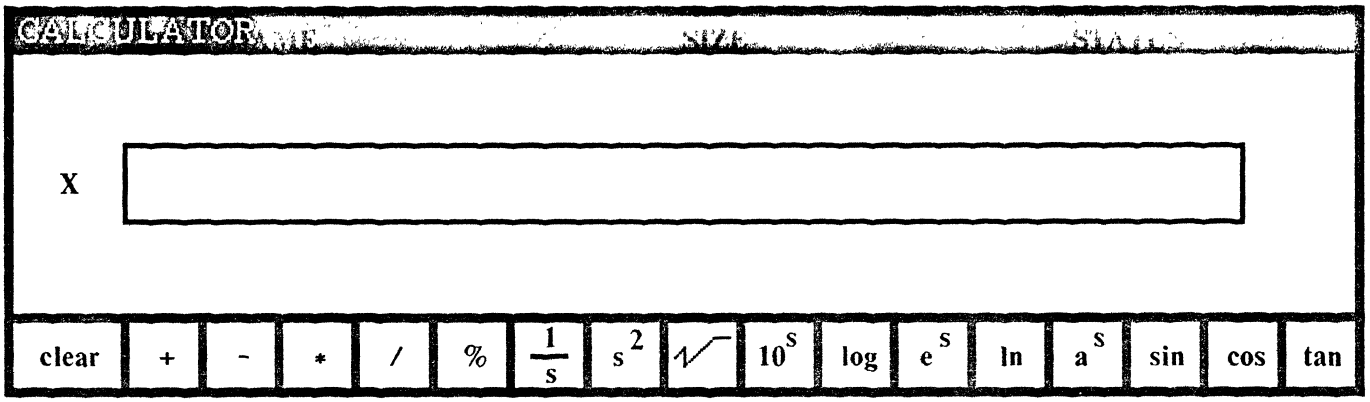
The RECORDER Function Window

(Note: in addition to these commands, the menu will also contain the standard window commands.)

Figure 1



CALCULATOR icon



The CALCULATOR Function Window

(Note: in addition to these commands, the menu will also contain the standard window commands.)

Figure 2