Actions, Tests and Flags in the Adventure Game Interpreter

********** ACTIONS **********

return()
     Stop scanning the current logics and return to the calling logics.

[**********] Var actions
Since vars are bytes, addition and subtraction may on occasion produce surprising results. For example, 128 + 128 = 0 and 128 - 150 = 234 when using vars. Test the values of vars before doing arithmetic when this sort of situation might arise.

increment(VAR)
     [Written as ++var]
     Increments the var, but not past 255.

decrement(VAR)
     [Written as --var]
     Decrements the var, but not past 0.

assignn(VAR, NUM)
     [Written as var = number]
     Set flag to number.

assignv(VAR, VAR)
     [Written as var1 = var2]
     Set var1 to var2.

addn(VAR, NUM)
     [Written as var += n]
     Add n to var.

addv(VAR, VAR)
     [Written as var1 += var2]
     Add var2 to var1.

subn(VAR, NUM)
     [Written as var -= n]
     Subtract n from var.

subv(VAR, VAR)
     [Written as var1 -= var2]
     Subtract var2 from var1.

lindirectv(VAR, VAR)
     [Written as var1 @= var2]

Left indirect assignment of vars. Takes the value in var1 and uses it as the var number in which to store the value of var2. In essence, var1 is a 'pointer' to the var for the assignment. Thus,

```
%var      var1      39
%var      var2      56
var1 = 27;
var2 = 14;
var1 @= var2;
```

would set variable 27 to 14.

lindirectn(VAR, NUM)
[Written as   var @= n]
As in lindirectv(), but assigns a number to the indirect location.

rindirect(VAR, VAR)
[Written as   var1 =@ var2]
Right indirect assignment of vars. Takes the value in var2 and uses it as the var number from which to get the value to store in var1. In this case, var2 is a pointer.

[**********] Flag actions

set(FLAG)
Set flag to 1.

reset(FLAG)
Set flag to 0.

toggle(FLAG)
Toggles the low bit of flag.

set.v(VAR)
Sets the flag whose number is in the var.

reset.v(VAR)
Resets the flag whose number is in the var.

toggle.v(VAR)
Toggles the flag whose number is in the var.

[**********] - Logic actions

new.room(NUM)
Throws out all loaded modules other than rm.0 and view.0, unanimates

all objects, sets the horizon to the default, then does a stop.sound(), user.control(), and unblock(). After loading the new room's logics it positions ego at the appropriate edge of the screen, animates him, loads whichever view ego had in the previous room, and sets his view to that. It then sets the flag init.log and begins scanning room 0.

new.room.f(VAR)
As in new.room(), but the room number is taken from the var.

load.logics(NUM)
Load the logics given by the number.

load.logics.f(VAR)
Load the logics whose number is in the var.

call(NUM)
Scan the logics given by the number. If the logics are not currently loaded, load them, scan them, then discard the logics. If the logics are not currently loaded, they should not load or animate anything, since discarding the logics will also release the memory allocated by those actions.

call.f(VAR)
As in call(), but the logic's number is taken from the var.

[**********] Picture actions

load.pic(VAR)
Load the picture whose number is in the var.

draw.pic(VAR)
Draw the picture whose number is in the var in the background screen. Note that this no longer affects the foreground screen. Call show.pic() when you're ready to show the user what you have wrought.

show.pic()
Bring the current background screen to the foreground.

discard.pic(VAR)
Release the memory allocated to the picture codes for the picture whose number is in var. This will also release all memory allocated by loads done after the load.pic(), so be careful.

overlay.pic(VAR)
Draw the picture whose number is in var over the current picture. It is unlikely that any color-fills will work, since fill only works on white and the picture being overlayed is probably not all white. As in draw.pic(), a call to show.pic() is necessary to bring the

new picture to the foreground screen.

[**********] View actions

load.view(VIEW)
     Load the view whose number is given by the number.

load.view.f(var)
     Load the view whose number is in the var.

discard.view(VIEW)
     Release the memory used by the view's description. As in discard.pic(), be careful since any loads done after the load of the view will also be released.

[**********] Animated object actions

animate.obj(OBJECT)
     Animate the given object. This will allocate some memory for saving the background of the object when it is drawn and will mark the object as ready for drawing/animation. It also does the equivalent of start.update(), start.motion(), start.cycling(), normal.cycle(), observe.blocks(), observe.horizon(), on.anything(), release.priority(), release.loop(), and observe.objects() on the object and sets its direction to 0.

unanimate.all()
     Unanimate all animated objects. If you're going to draw a second picture in a room (using draw.pic() without doing a new.room()) do this first to make sure that you don't end up with a number of animated objects which you don't want in the new picture. You'll have to do an animate.obj(), draw(), stop.update(), etc. on any object which you want in the new picture. The object will have it's same view, loop, cel, and position unless you change them (set.view(), etc. won't be necessary). Remember that all the things that animate.obj() does will be done when you re-animate.

draw(OBJECT)
     Draw the object in the background screen and, if a show.pic() has been done since the last draw.pic() or overlay.pic(), put the object in the foreground screen as well.

erase(OBJECT)
     Erase the object from the screen. Erased objects do not move or interact with drawn objects.

position(OBJECT, NUM, NUM)
     Parameters: object, x coord, y coord.
     Set the position of the object (the lower left corner of its baseline) to the (x, y) coords given in the numbers. x is measured from the right edge of the screen, y from the top. If part of the object will be off the screen, if the object is above the horizon and must observe it, or if the object's baseline is on a priority line which it must observe, the object is repositioned. Doing a position() of a drawn object will leave a clone behind where the objects was. After drawing, use reposition().

position.f(OBJECT, VAR, VAR)
     Parameters: object, x coord, y coord.
     As in position(), but the position is taken from vars.

get.posn(OBJECT, VAR, VAR)
     Parameters: object, x position, y position.
     Return the (x, y) position of the object in the variables.

reposition(OBJECT, VAR, VAR)
     Parameters: object, x displacement, y displacement.
     Repositions the object by the delta-x and delta-y in the vars. The high bit of the var's byte is the sign bit. Arithmetic operations on vars or assignment of a negative number to a var will set the sign bit properly. This operation should done on objects after they have been drawn. Before drawing use position() or position.f().

[**********]   Views of animated objects

set.view(OBJECT, VIEW)
     Set the view of the object to view. If the previous loop or cel number of the object is greater than the number of loops or cels in the current view, sets the offending loop or cel number to 0. Otherwise, loop and cel remain the same.

set.view.f(OBJECT, VAR)
     Set the view of the object to the view number in the var.

set.loop(OBJECT, NUM)
     Set the loop of the object to that in the number. Loop numbers and the corresponding direction in which the object's view faces are:
          0   faces right
          1   faces left
          2   faces front
          3   faces back
     Loops are automatically set to that corresponding to an object's current direction. A non-moving object (direction = 0) remains in its last loop.

set.loop.f(OBJECT, VAR)
     Set the object's loop to that in the var.

fix.loop(OBJECT)
     Fix the object's loop. It will no longer adjust to the direction
     in which the object is moving.

release.loop(OBJECT)
     Undo a fix.loop(). The object will now face in the direction
     appropriate to its direction.

set.cel(OBJECT, NUM)
     Set the object's cel to that in the number. Cels are set automatically
     when the object is cycling. An object which is not cycling remains in
     the last set cel.

set.cel.f(OBJECT, VAR)
     Set the object's cel to that in the var.

last.cel(OBJECT, VAR)
     Return the cel number of the last cel of the current loop of the
     object's current view in var.

current.cel(OBJECT, VAR)
     Return the current cel number of the object in var.

current.loop(OBJECT, VAR)
     Return the current loop number of the object in var.

current.view(OBJECT, VAR)
     Return the current view of the object in var.

number.of.loops(OBJECT, VAR)
     Return the number of loops in the current view of the object in var.

[**********]   Priority control of animated objects

set.priority(OBJECT, NUM)
     Set the priority of the object to num. This fixes the priority of
     the object.

set.priority.f(OBJECT, VAR)
     Set the priority of the object to that in the var.

release.priority(OBJECT)

Free the priority of the object so that it is set automatically depending on the object's y position. This is done by animate.obj().

get.priority(OBJECT, VAR)
Return the current priority of the object in the var.

[**********] Attributes of animated objects

stop.update(OBJECT)
Stop the object from updating. The object will remain on the screen, but will no longer move or cycle. The object's image will not be updated during the animation cycle. This should be done when possible to reduce the animation load on the interpreter and keep it from running too slow.

start.update(OBJECT)
Start updating a stop.update() object. This is done by animate.obj().

force.update(OBJECT)
Force an update of a stop.update() object, but don't start.update() it. This can be done to force a set.view(), set.loop(), or set.cel() to show on the screen without actually starting update on the object.

ignore.horizon(OBJECT)
Let the object ignore the horizon, so that it can move above or be positioned above the horizon.

observe.horizon(OBJECT)
Force the object to observe the horizon. If this is done on an object which is above the horizon, the object will be repositioned below the horizon. This is done by animate.obj().

set.horizon(NUM)
Set the y coordinate of the horizon to the number. The default horizon, set by new.room(), is 36.

object.on.water(OBJECT)
Force the object to remain on water. This requires that the entire baseline of the object be on water priority.

object.on.land(OBJECT)
Force the object to stay off water. This requires that none of the baseline of the object be on water priority.

object.on.anything(OBJECT)
Let the object go anywhere. This undoes object.on.water() and object.on.land(). This is done by animate.obj().

ignore.objs(OBJECT)
     Lets the object's baseline pass through the baselines of other objects.

observe.objs(OBJECT)
     Tells the object to stop if it's baseline hits that of another object.
     This is done by animate.obj().

distance(OBJECT, OBJECT, VAR)
     Returns the distance between the centers of the baselines of the
     two objects in var. If one or both of the objects is not drawn,
     this returns max.flag.value (= 255).

[**********]   Cycling of animated objects

stop.cycling(OBJECT)
     Stop the automatic cycling through cels for the object.

start.cycling(OBJECT)
     Start the automatic cycling through cels for the object. This is done
     by animate.obj().

normal.cycle(OBJECT)
     Cycle the object from the current cel number to high cel number.
     When the last cel is reached, start again at cel 0. This is done by
     animate.obj().

end.of.loop(OBJECT, FLAG)
     Reset the flag. Increment the cel number at each animation cycle.
     When the last cel of the loop is reached, stop cycling and set the flag.

reverse.cycle(OBJECT)
     Cycle the object from the current cel number to cel 0. When cel 0
     is reached, start again at the last cel of the current loop.

reverse.loop(OBJECT, FLAG)
     Reset the flag. Decrement the cel number at each animation cycle.
     When cel 0 is reached, stop cycling and set the flag.

cycle.time(OBJECT, VAR)
     Set the cycle frequency of the object (the number of animation cycles
     between cycling the object's cels) to the number in the var. Reset
     to 1 by new.room().

[**********]   Motion of animated objects

stop.motion(OBJECT)
     Prevent the object from moving, though cycling will continue.

start.motion(OBJECT)
     Allow an object to move. This is done by animate.obj().

step.size(OBJECT, VAR)
     Set the step size of the object (how far it moves in each animation
     cycle) to the number in var. Reset to 1 by new.room().

step.time(OBJECT, VAR)
     Set the step frequency of the object (the number of animation cycles
     betweeen moves of the object) to the number in var. Reset to 1 by
     new.room().

move.obj(OBJECT, NUM, NUM, NUM, FLAG)
     Parameters: object, x coord, y coord, step size, flag.
     Reset the flag, then start moving the object to the given (x, y)
     position. Change the appropriate coordinates by the step size at
     each animation interval. If the step size parameter is zero
     (the normal case), the motion uses the default step size of the object.
     When the object gets within the step size of the destination, stop
     the motion and set the flag.

move.obj.f(OBJECT, VAR, VAR, VAR, FLAG)
     Parameters: object, x coord, y coord, step size, flag.
     Like move.obj(), destination coordinates and step size are passed in
     vars.

follow.ego(OBJECT, NUM, FLAG)
     Parameters: object, distance for collision, flag.
     Reset the flag and start the object following ego with its current
     stepsize. When the distance to the object is less than the greater
     of the object's stepsize and the specified distance for collision,
     stop the object's motion and set the flag.

wander(OBJECT)
     Start the object wandering in random directions for random distances.

normal.motion(OBJECT)
     Undoes a previous follow.ego(), move.obj(), or wander() before it is
     completed.

set.dir(OBJECT, VAR)
     Set the direction of the object to that in the var. Note that this
     will not work for ego -- set the var ego.dir to change ego's direction.

get.dir(OBJECT, VAR)
     Get the current direction of the object in the variable.

[**********] Block actions

ignore.blocks(OBJECT)
     Let an object priority 1 and blocks set by block().

observe.blocks(OBJECT)
     Require the object to observe priority 1 and blocks set by block.
     This is done by animate.obj().

block(NUM, NUM, NUM, NUM)
     Parameters: upper left x, upper left y, lower right x, lower right y.
     Set a block to stop object motion. Only one block at a time may be
     set.
****** How about a number of blocks?

unblock()
     Remove the block set by block().

[**********] Inventory object actions

get(OBJECT)
     Add the object to ego's inventory.

getf(VAR)
     Add the object whose number is in var to ego's inventory. Generally
     used for debugging, not in the game.

drop(OBJECT)
     Remove the object from ego's inventory. The object is now gone forever.

put(OBJECT, VAR)
     Put the number in var into the room number field of the inventory object.

put.f(VAR, VAR)
     Parameters: inventory object, room number.
     Like put(), inventory object and room number are contained in variables.

get.room.f(VAR, VAR)
     Parameters: inventory object, var number.
     Get the room number field of inventory object in the variable.

[**********] Sounds

load.sound(NUM)

Load the sound.

sound(NUM, FLAG)
Reset the flag, then play the sound. When the sound is finished, set the flag. If sounds are off, the flag is set immediately -- therefore don't use this to time anything but the sound.

stop.sound()
Stop the current sound from playing. Sets the end flag for the sound.

[**********] Message/text display

Messages are strings of fewer than 255 characters which may contain the following special commands:

\	Take the next character (except '\n' below) literally
\n	Begin a new line
%wn	Include word number n from the parsed line (1 <= n <= 255)
%sn	Include string number n (0 <= n <= 255)
%mn	Include message number n from this room (0 <= n <= 255)
%gn	Include global message number n from room 0 (0 <= n <= 255)
%vn|m	Print the value of var #n. If the optional '|m' is present, print in a field of width m with leading zeros.

Thus,

```
%message  1    "This is %m2"
%message  2    "message 1."
print(1);
```

will print as

```
This is message 1.
```

If var number 5 is 23,

```
%message  1    "The value is: %v5|3."
```

will print as

```
The value is 023.
```

The '\' escape character is interpreted by both the compiler and the interpreter, so it is sometimes necessary to do a lot of \\\\ing to get things right. The following table should help:

Char which you really want	Use
"	\"
%	\\%
\	\\\\

All spaces in messages are now significant (multiple spaces used to be compressed to one). This will require a change in the way messages are entered. Instead of

```
%message  1    "blah blah blah
               blah blah blah"
```

use

```
%message  1
"blah blah blah
 blah blah blah"
```

print(MSGNUM)
Print the message in a pop-up window. See the discussion of message formats below.

print.f(VAR)
Print the message whose number is in the var in a pop-up window.

display(NUM, NUM, MSGNUM)
Parameters: row, column, message number.
Print the message at the (row, col) position given by the numbers.

display.f(VAR, VAR, VAR)
Parameters: row, column, message number.
Print the message whose number is in the last var at the (row, col) position given by the first two vars.

clear.lines(NUM, NUM, NUM)
Parameters: top row, bottom row, screen attribute.
Clear the screen rows (inclusive) between the top and bottom rows to the given screen attribute. See set.text.attribute() for the screen attributes.

text.screen()
Go to a text screen. On machines with a special hardware text screen this will generally give clearer, faster text displays.

graphics()
Return to the graphics screen after a text.screen() call.

set.cursor.char(MSGNUM)
Set the cursor character to the first character of the message. At startup, there is no cursor.

set.text.attribute(NUM, NUM)

Parameters: foreground color, background color.
Set the foreground and background colors for text. Not all combinations will necessarily be supported on all machines. We will try to approximate where we can, but no guarantees. The colors are:

0	black
1	dark blue
2	dark green
3	cyan
4	red
5	magenta
6	brown
7	light grey
8	dark grey
9	light blue
10	light green
11	light cyan
12	pink
13	light magenta
14	yellow
15	white

All combinations will (I believe) be supported on the Atari ST, Amiga, and the NEC 9801.

On the Apple, if the background is anything but black the text will be inverse, otherwise it will be normal.

On the IBM, all colors will be supported in text mode. In graphics mode, foreground colors 0-7 will be supported on the PCjr & the EGA, but will be mapped into black, cyan, magenta, and white on the PC. If the background color is anything but black in graphics mode, the text will print in inverse (black on white).

shake.screen(NUM)
Shake the screen quickly in a set of figure eights. The number is the number of figure eights to do. This will be a no-op on some hardware, so don't depend on it.

[**********] Screen handling

configure.screen(NUM, NUM, NUM)
Parameters: picture row, input row, status row.
This call, which should be done as soon as the game starts, sets where the various components of the screen are placed. The first number is the CHARACTER row number (starting at zero) for the upper left corner of the picture, the second is the row number for the

input line, and the third is the row number for the status line.

status.line.on()
Turn the status line on. This displays an inverse line which shows the score and the state of the sound toggle.

status.line.off()
Turns the status line off and clears it to black. The status line is off at startup.

[**********] String handling

There are 10 strings (1 - 10) of up to 19 characters which can be used in messages. String # 0 is the prompt.

set.string(NUM, MSGNUM)
Parameters: string number, message number.
Copy the message into the string given by the number.

get.string(NUM, MSGNUM, NUM, NUM)
Parameters: string number, message number, row, column.

Prints the message as a prompt at the given screen position, then allows the user to enter the string for string number NUM. If the row is >24, the message will be printed at the current cursor position.

Since string 0 is the prompt, set the prompt by
```
%message 1 "> "
set.string(0, 1);
```

The code to let the user set the prompt is
```
%message 2 "New prompt: "
get.string(0, 2);
```

If the user presses ESC, nothing will be copied to the string, so it is advisable to set a default with set.string() before calling get.string().

word.to.string(NUM, NUM)
Parameters: word number, string number.
Copy the word into the string.

parse(NUM)
Parse the given string as if it were a normal input line from the user. 'Have.input' will be set, as will 'unknown.word' if applicable. The words will be available to all said() tests for the remainder of the current logic scan.

get.num(MSGNUM, VAR)
     Prompt the user with the message and get a (purportedly) numeric reply.
     Put the number into var. If a non-numeric reply is typed, var will
     be 0.

[**********]   Input handling

prevent.input()
     Clear the input line and do not accept input from the user. Input
     is off at startup.

accept.input()
     Display the input line and accept input from the user.

set.key(NUM, NUM, NUM)
     Parameters: low byte of keycode, high byte of keycode, controller number.
     Assign a key to a controller. The first number is the low byte
     of the key's keycode, the second is the high byte. The last number
     is the number of the controller which is to be activated by this
     key. The keycodes are given in the IBM Tech Ref. in section 2 under
     "Keyboard Encoding and Useage", and (I think) in the BASIC manual.

     ASCII characters have a zero high byte (second number) and the ASCII
     code in the low byte. Function keys, Alt keys, etc. have the a
     zero low byte (first number) and a high byte which is the extended
     keycode from the manual. Joystick buttons have the following
     keycodes (these are in 'sysdefs'):

                                        low byte   high byte
          single click, button 0   1   1
          single click, button 1   1   2
          double click, button 0   1   3
          double click, button 1   1   4

     The following are the set.key() commands to give the keyboard
     map of Black Cauldron:

          set.key(0, 59, c.sound.toggle)   [F1
          set.key(19, 0, c.sound.toggle);   [^S
          set.key(0, 60, c.help);   [F2
          set.key(0, 61, c.save.game);   [F3
          set.key(0, 62, c.useit);   [F4
          set.key(0, 63, c.restore.game);   [F5
          set.key(0, 64, c.doit);   [F6
          set.key(0, 65, c.restart);   [F7
          set.key(0, 66, c.lookit);   [F8

```
          set.key(0, 68, c.show.mem);        [F10
          set.key(9, 0, c.status);           [TAB
          set.key(0, 32, c.debug);           [@D
          set.key(10, 0, c.reset.joy);       [^J
          set.key(3, 0, c.cancel.line);      [^C
          set.key(5, 0, c.echo.line);        [^E
          set.key(27, 0, c.pause);           [ESC
          set.key(18, 0, c.rgb.toggle);      [^R
          set.key(16, 0, c.new.prompt);      [^P

          set.key(joy.low.byte, button0, c.doit);
          set.key(joy.low.byte, button1, c.useit);
          set.key(joy.low.byte, button0.dbl, c.lookit);
          set.key(joy.low.byte, button1.dbl, c.status);
```

The controller definitions are in 'sysdefs'. You may map up to 29 keys. More than one key may map to a single controller, but a single key can't map to more than one controller.

****** Note that the keycodes used here are machine-dependent. This
****** code should be kept in one module if possible to minimize
****** re-write for new machines.

[**********] Add to picture

add.to.pic(VIEW, NUM, NUM, NUM, NUM, NUM, NUM)

Parameters: view, loop, cel, x pos, y pos, object priority, box priority.
Draw the view in the picture without saving its background. If the object priority parameter is 0, the object's priority is that of the priority band in which it is placed. The object cannot be animated and cannot be erased except by drawing something over it. The object is added to the picture with a box of 'box priority' which extends from its baseline to the bottom of the next lowest priority band.
If this is 0, it prevents other objects from 'popping' through it.
Box priorities of 4 and above do not add any box. Add.to.picture() ignores all priority lines, object baselines, and block() commands -- it can go anywhere in the picture.

add.to.picture.f(VAR, VAR, VAR, VAR, VAR, VAR, VAR)

Parameters: view, loop, cel, x pos, y pos, object priority, box priority.
Same as above, but from variables.

[**********] User requested actions

status()

Goes to an inverse text screen and displays the player's score and a list of what he is carrying.

save.game()
Prompt the player for a letter between 'a' and 'z' under which to save the current game, then save it.

restore.game()
Prompt the player for the letter under which a game was saved, then restore that game.

init.disk()
Initialize a disk for saving games on. Special signatures are written on the disk for identification so that we don't try to save on a player's game disk.

restart.game()
Restart the game. Room 0 logics, view 0, and words.tok remain in memory, everything else is reloaded and flags and variables are reset. Strings are not affected. The 'restart.in.progress' flag is set, in case you need to know about this event.

[**********] Show object view

show.obj(VIEW)
Show a special view of an object and print a description of it. Used for response to 'look at object'.

[**********] Miscellaneous

random(NUM, NUM, VAR)
Parameters: minimum value, maximum value, variable.
Return, in the variable, a random number between the minimum and maximum values, inclusive. This uses a linear congruential generator seeded from the system time, and so should be fairly good. Tests show that the distribution is VERY uniform.

program.control()
Assume program control of ego, so that ego does not move in the direction indicated by the user. This is not normally necessary, as move.obj(ego) and wander(ego) do an implicit program.control(). The benefit (if any) of this over doing a stop.motion(ego) and a stop.cycling(ego) is that the variable ego.dir is not affected by the user's input as it would be with the stops.

player.control()
Return control of ego to the player. This is the only way to terminate

a wander(ego), and is the way to terminate a move.obj(ego) prematurely. In the latter case, the end flag is not set.

obj.status.f(VAR)
Display the state of an object in a pop-up window. Used for debugging; currently displays x position, y position, and priority.

quit()
Terminate the game. On direct I/O versions of the game, your're hung. On DOS versions, returns you to DOS.

show.mem()
Show the total heap space available, how much is currently being used, and the amount of unused stack.

pause()
Pause the game. Displays a message to the effect that the game is paused in a pop-up window. Waits for the user to press ENTER or ESC before continuing.

echo.line()
Echo the previous player input to the input line. The echo starts from the current position in the line.

cancel.line()
Cancel the current input line.

init.joy()
Prompt the user to center his joystick and recalibrate the joystick routines.

toggle.sound()
Toggle the on/off state of the sound.

toggle.monitor()
Toggle between the composite 16 color mode and 4 color RGB mode. No effect on a PCjr, Tandy 1000, or a PC with an EGA.
****** This is pretty machine-dependent. What to do?

version()
Display the version message of the interpreter in a pop-up window. Used for debugging.

script.size(NUM)
Sets the script buffer size. Default size is 50 events. Do this in the current.room == 0 logics of room.0, before a new.room() is done. Maximum script buffer events used is displayed in the show.mem() display.

max.drawn(NUM)

Sets the maximum number of animated objects which can be drawn at a time. Default is 15 animated objects. Do this at the same time as script.size().

********** TESTS **********

equaln(VAR, NUM)
     [Written as   var == num]
     True if the var has the value num. A special case is the test
          if (var)
     which is expanded to
          if (var == 0)

equalv(VAR, VAR)
     [Written as   var1 = var2]
     True if var1 has the same value as var2.

lessn(VAR, NUM)
     [Written as   var < num]
     True if the value of var is less than num.

lessv(VAR, VAR)
     [Written as   var1 < var2]
     True if the value of var1 is less than the value of var2.

greatern(VAR, NUM)
     [Written as   var > num]
     True if the value of var is greater than num.

greaterv(VAR, NUM)
     [Written as   var1 > var2]
     True if the value of var1 is greater than the value of var2.

isset(FLAG)
     [Written as   flag]
     True if the flag is set.

isset.v(VAR)
     True if the flag whose number is the value of the var is set.

has(OBJECT)
     True if the object is in ego's inventory.

obj.in.room(OBJECT, VAR)
     True if the object is in the room whose number is in var.

posn(OBJECT, NUM, NUM, NUM, NUM)
     Parameters: object, upper left x, upper left y, lower right x, lower right y
     True if the object is in the rectangle described by the points
     (including the coordinates given).

controller(NUM)
     True if the given controller is set.

have.key()
     True if a key is waiting to be read. Note that if a 'prevent.input()'
     has not been done, the likelyhood of this being true is minimal --
     virtually all input will go to the input line.

said(WORDLIST)
     True if the number of non-ignored words in the input line is the
     same as that in the word list and the non-ignored words in the input
     match, in order, the words in the word list. The special word
     'anyword' (or whatever is defined as word 1 in 'words.txt') in the
     word list matches any non-ignored word in the input.

********** VARS **********

current.room
Contains the current room number.

previous.room
Contains the number of the previous room.

edge.ego.hit
If non-zero, indicates that ego hit an edge of the picture. The edge hit is given by:

top	1
right	2
bottom	3
left	4

score
Contains the player's current score. This is modified by the logics and printed on the status line or the status page.

obj.hit.edge
If non-zero, indicates that an object (other than ego) hit the edge of the picture. The value of the var is the number of the object.

edge.obj.hit
When non-zero, indicates the edge of the screen that the object hit.

ego.dir
Gives ego's current direction. This is the direction provided by user -- a move.obj() will not affect it. The directions and their corresponding values are

stopped	0
up	1
up & right	2
right	3
down & right	4
down	5
down & left	6
left	7
up & left	8

max.score
This is the maximum score attainable in the game. It's only purpose is to allow the "You have x points out of a possible y" message on the status screen.

memory.left

The number of pages (512 byte blocks) of memory remaining free in the heap.

unknown.word
This flag is set to the number of any unknown word in the user's input, or to zero if all the words were recognized. To handle unknown words, use the following code before any said() tests in room 0:

```
%message  1    "I don't understand \"%1\""
%message  2    "I don't understand \"%2\""
%message  3    "I don't understand \"%3\""
if (have.input && unknown.word) {
     reset(have.input);
     print.f(unknown.word);
     }
```

animation.interval
This is the number of timer ticks (1/18.2 secs) between animation cycles and logic scans.

elapsed.seconds
elapsed.minutes
elapsed.hours
elapsed.days
These contain the play time since the user started the game. Restored games restore these vars. This is the most accurate time-base for any real-time needs. Incrementing or decrementing flags is not accurate if animation.interval is 0 or if animation over-runs timer interrupts.

double.click.delay

current.ego
Contains the view number of ego's current view.

error.number
error.parameter

machine.type

********** FLAGS **********

The following flags are set when:

on.water
: Ego's baseline is entirely on water.

ego.hidden
: No pixels of ego are visible on the screen.

have.input
: The user has pressed RETURN after typing text. Input has been parsed for said() tests.

hit.special
: At least one pixel of ego's baseline is on special priority.

have.match
: A said() test has produced a match. No more said() tests will succeed unless have.match is reset.

init.log
: All logics should do their initializations.

restart.in.progress
: This is set through the first logic scan following a restart of the game. Don't bother putting up banner screen, etc. when this is set.

no.script
: Setting this flag will keep the interpreter from adding subsequent scriptable actions to the script buffer.

The script refered to above is a list of the actions which have brought the game to the current state. It is used for restoring the game. The actions which add to the script are:

```
load.logic()
load.view()
load.pic()
load.sound();
draw.pic();
overlay.pic();
add.to.pic();       (this takes 4 script entries)
discard.pic();
discard.view();
```

```
     and any 'var' forms of the above.

forget.add.to.pic

enable.double.click

sound.on
```