

# **ZIL Reference Guide**

*Henrik Åsman*

# Table of Contents

<u>ZIL Reference Guide</u>	<u>12</u>
<u>Introduction</u>	<u>12</u>
<u>Syntax</u>	<u>12</u>
<u>Regarding TRUE and FALSE</u>	<u>13</u>
<u>MDL builtins and ZIL library (use outside ROUTINE)</u>	<u>14</u>
<u>* (multiply)</u>	<u>14</u>
<u>+ (add)</u>	<u>14</u>
<u>- (subtract)</u>	<u>14</u>
<u>/ (divide)</u>	<u>14</u>
<u>0?</u>	<u>15</u>
<u>1?</u>	<u>15</u>
<u>==?</u>	<u>15</u>
<u>=?</u>	<u>15</u>
<u>ADD-TELL-TOKENS</u>	<u>16</u>
<u>ADD-WORD</u>	<u>16</u>
<u>ADJ-SYNONYM</u>	<u>17</u>
<u>AGAIN</u>	<u>17</u>
<u>ALLTYPES</u>	<u>17</u>
<u>AND</u>	<u>18</u>
<u>AND?</u>	<u>18</u>
<u>ANDB</u>	<u>18</u>
<u>APPLICABLE?</u>	<u>18</u>
<u>APPLY</u>	<u>19</u>
<u>APPLYTYPE</u>	<u>19</u>
<u>ASCII</u>	<u>19</u>
<u>ASSIGNED?</u>	<u>20</u>
<u>ASSOCIATIONS</u>	<u>20</u>
<u>ATOM</u>	<u>20</u>
<u>AVALUE</u>	<u>20</u>
<u>BACK</u>	<u>20</u>
<u>BIND</u>	<u>20</u>
<u>BIT-SYNONYM</u>	<u>21</u>
<u>BLOCK</u>	<u>21</u>
<u>BOUND?</u>	<u>21</u>
<u>BUZZ</u>	<u>22</u>
<u>BYTE</u>	<u>22</u>
<u>CHECK-VERSION?</u>	<u>22</u>
<u>CHRSET</u>	<u>22</u>

<u>CHTYPE</u>	<u>22</u>
<u>CLOSE</u>	<u>23</u>
<u>COMPILATION-FLAG</u>	<u>23</u>
<u>COMPILATION-FLAG-DEFAULT</u>	<u>23</u>
<u>COMPILATION-FLAG-VALUE</u>	<u>23</u>
<u>COND</u>	<u>23</u>
<u>CONS</u>	<u>23</u>
<u>CONSTANT</u>	<u>23</u>
<u>CRLF</u>	<u>24</u>
<u>DECL-CHECK</u>	<u>24</u>
<u>DECL?</u>	<u>24</u>
<u>DEFAULT-DEFINITION</u>	<u>24</u>
<u>DEFINE</u>	<u>25</u>
<u>DEFINE-GLOBALS</u>	<u>26</u>
<u>DEFINE20</u>	<u>26</u>
<u>DEFINITIONS</u>	<u>26</u>
<u>DEFMAC</u>	<u>26</u>
<u>DEFSTRUCT</u>	<u>26</u>
<u>DELAY-DEFINITION</u>	<u>26</u>
<u>DIR-SYNONYM</u>	<u>26</u>
<u>DIRECTIONS</u>	<u>27</u>
<u>EMPTY?</u>	<u>27</u>
<u>END-DEFINITIONS</u>	<u>27</u>
<u>ENDBLOCK</u>	<u>27</u>
<u>ENDPACKAGE</u>	<u>27</u>
<u>ENDSECTION</u>	<u>27</u>
<u>ENTRY</u>	<u>27</u>
<u>EQVB</u>	<u>27</u>
<u>ERROR</u>	<u>27</u>
<u>EVAL</u>	<u>28</u>
<u>EVALTYPE</u>	<u>28</u>
<u>EXPAND</u>	<u>28</u>
<u>FILE-FLAGS</u>	<u>28</u>
<u>FILE-LENGTH</u>	<u>28</u>
<u>FLOAD</u>	<u>28</u>
<u>FORM</u>	<u>29</u>
<u>FUNCTION</u>	<u>29</u>
<u>FUNNY-GLOBALS?</u>	<u>30</u>
<u>G=?</u>	<u>30</u>
<u>G?</u>	<u>31</u>

<u>GASSIGNED?</u>	31
<u>GBOUND?</u>	31
<u>GC</u>	31
<u>GDECL</u>	32
<u>GET-DECL</u>	32
<u>GETB</u>	32
<u>GETPROP</u>	32
<u>GLOBAL</u>	32
<u>GROW</u>	32
<u>GUNASSIGN</u>	32
<u>GVAL</u>	33
<u>IFFLAG</u>	33
<u>ILIST</u>	33
<u>IMAGE</u>	34
<u>INCLUDE</u>	34
<u>INCLUDE-WHEN</u>	34
<u>INDENT-TO</u>	34
<u>INDEX</u>	34
<u>INDICATOR</u>	34
<u>INSERT</u>	34
<u>INSERT-FILE</u>	34
<u>ISTRING</u>	34
<u>ITABLE</u>	35
<u>ITEM</u>	36
<u>IVECTOR</u>	36
<u>L=?</u>	36
<u>L?</u>	36
<u>LANGUAGE</u>	36
<u>LEGAL?</u>	37
<u>LENGTH</u>	37
<u>LENGTH?</u>	37
<u>LINK</u>	37
<u>LIST</u>	37
<u>LONG-WORDS?</u>	38
<u>LOOKUP</u>	38
<u>LPARSE</u>	38
<u>LSH</u>	38
<u>LTABLE</u>	39
<u>LVAL</u>	39
<u>M-HPOS</u>	39

<u>MAPF</u>	<u>39</u>
<u>MAPLEAVE</u>	<u>39</u>
<u>MAPR</u>	<u>39</u>
<u>MAPRET</u>	<u>39</u>
<u>MAPSTOP</u>	<u>39</u>
<u>MAX</u>	<u>39</u>
<u>MEMBER</u>	<u>40</u>
<u>MEMQ</u>	<u>40</u>
<u>MIN</u>	<u>40</u>
<u>MOBLIST</u>	<u>41</u>
<u>MOD</u>	<u>41</u>
<u>MSETG</u>	<u>41</u>
<u>N==?</u>	<u>41</u>
<u>N=?</u>	<u>41</u>
<u>NEW-ADD-WORD</u>	<u>42</u>
<u>NEWTYPE</u>	<u>42</u>
<u>NEXT</u>	<u>42</u>
<u>NOT</u>	<u>42</u>
<u>NTH</u>	<u>42</u>
<u>OBJECT</u>	<u>43</u>
<u>OBLIST?</u>	<u>43</u>
<u>OFFSET</u>	<u>43</u>
<u>OPEN</u>	<u>43</u>
<u>OR</u>	<u>43</u>
<u>OR?</u>	<u>43</u>
<u>ORB</u>	<u>44</u>
<u>ORDER-FLAGS?</u>	<u>44</u>
<u>ORDER-OBJECTS?</u>	<u>44</u>
<u>ORDER-TREE?</u>	<u>45</u>
<u>PACKAGE</u>	<u>45</u>
<u>PARSE</u>	<u>45</u>
<u>PLTABLE</u>	<u>45</u>
<u>PNAME</u>	<u>46</u>
<u>PREP-SYNONYM</u>	<u>46</u>
<u>PRIMTYPE</u>	<u>46</u>
<u>PRIN1</u>	<u>46</u>
<u>PRINC</u>	<u>46</u>
<u>PRINT</u>	<u>47</u>
<u>PRINT-MANY</u>	<u>47</u>
<u>PRINTTYPE</u>	<u>47</u>

<a href="#"><u>PROG</u></a>	<a href="#"><u>47</u></a>
<a href="#"><u>PROPDEF</u></a>	<a href="#"><u>48</u></a>
<a href="#"><u>PTABLE</u></a>	<a href="#"><u>48</u></a>
<a href="#"><u>PUT</u></a>	<a href="#"><u>48</u></a>
<a href="#"><u>PUT-DECL</u></a>	<a href="#"><u>48</u></a>
<a href="#"><u>PUTB</u></a>	<a href="#"><u>48</u></a>
<a href="#"><u>PUTPROP</u></a>	<a href="#"><u>49</u></a>
<a href="#"><u>PUTREST</u></a>	<a href="#"><u>49</u></a>
<a href="#"><u>QUIT</u></a>	<a href="#"><u>49</u></a>
<a href="#"><u>QUOTE</u></a>	<a href="#"><u>49</u></a>
<a href="#"><u>READSTRING</u></a>	<a href="#"><u>49</u></a>
<a href="#"><u>REMOVE</u></a>	<a href="#"><u>49</u></a>
<a href="#"><u>RENTY</u></a>	<a href="#"><u>49</u></a>
<a href="#"><u>REPEAT</u></a>	<a href="#"><u>49</u></a>
<a href="#"><u>REPLACE-DEFINITION</u></a>	<a href="#"><u>50</u></a>
<a href="#"><u>REST</u></a>	<a href="#"><u>51</u></a>
<a href="#"><u>RETURN</u></a>	<a href="#"><u>51</u></a>
<a href="#"><u>ROOM</u></a>	<a href="#"><u>51</u></a>
<a href="#"><u>ROOT</u></a>	<a href="#"><u>51</u></a>
<a href="#"><u>ROUTINE</u></a>	<a href="#"><u>52</u></a>
<a href="#"><u>ROUTINE-FLAGS</u></a>	<a href="#"><u>52</u></a>
<a href="#"><u>SET</u></a>	<a href="#"><u>52</u></a>
<a href="#"><u>SET-DEFSTRUCT-FILE-DEFAULTS</u></a>	<a href="#"><u>52</u></a>
<a href="#"><u>SETG</u></a>	<a href="#"><u>52</u></a>
<a href="#"><u>SETG20</u></a>	<a href="#"><u>52</u></a>
<a href="#"><u>SORT</u></a>	<a href="#"><u>52</u></a>
<a href="#"><u>SPNAME</u></a>	<a href="#"><u>52</u></a>
<a href="#"><u>STRING</u></a>	<a href="#"><u>53</u></a>
<a href="#"><u>STRUCTURED?</u></a>	<a href="#"><u>53</u></a>
<a href="#"><u>SUBSTRUC</u></a>	<a href="#"><u>53</u></a>
<a href="#"><u>SYNONYM</u></a>	<a href="#"><u>54</u></a>
<a href="#"><u>SYNTAX</u></a>	<a href="#"><u>54</u></a>
<a href="#"><u>TABLE</u></a>	<a href="#"><u>54</u></a>
<a href="#"><u>TELL-TOKENS</u></a>	<a href="#"><u>55</u></a>
<a href="#"><u>TOP</u></a>	<a href="#"><u>55</u></a>
<a href="#"><u>TUPLE</u></a>	<a href="#"><u>56</u></a>
<a href="#"><u>TYPE</u></a>	<a href="#"><u>56</u></a>
<a href="#"><u>TYPE?</u></a>	<a href="#"><u>57</u></a>
<a href="#"><u>TYPEPRIM</u></a>	<a href="#"><u>57</u></a>
<a href="#"><u>UNASSIGN</u></a>	<a href="#"><u>57</u></a>

<u>UNPARSE</u>	<u>58</u>
<u>USE</u>	<u>58</u>
<u>USE-WHEN</u>	<u>58</u>
<u>VALID-TYPE?</u>	<u>58</u>
<u>VALUE</u>	<u>58</u>
<u>VECTOR</u>	<u>58</u>
<u>VERB-SYNONYM</u>	<u>59</u>
<u>VERSION</u>	<u>59</u>
<u>VERSION?</u>	<u>60</u>
<u>VOC</u>	<u>60</u>
<u>XORB</u>	<u>60</u>
<u>ZGET</u>	<u>60</u>
<u>ZIP-OPTIONS</u>	<u>61</u>
<u>ZPUT</u>	<u>61</u>
<u>ZREST</u>	<u>61</u>
<u>ZSTART</u>	<u>62</u>
<u>Z-code builtins (use inside ROUTINE)</u>	<u>63</u>
<u>*, MUL</u>	<u>63</u>
<u>+, ADD</u>	<u>63</u>
<u>-, SUB</u>	<u>63</u>
<u>/, DIV</u>	<u>64</u>
<u>0?, ZERO?</u>	<u>64</u>
<u>1?</u>	<u>64</u>
<u>=?, ==?, EQUAL?</u>	<u>64</u>
<u>AGAIN</u>	<u>65</u>
<u>AND</u>	<u>65</u>
<u>APPLY</u>	<u>66</u>
<u>ASH, ASHIFT</u>	<u>66</u>
<u>ASSIGNED?</u>	<u>67</u>
<u>BACK</u>	<u>67</u>
<u>BAND, ANDB</u>	<u>67</u>
<u>BCOM</u>	<u>68</u>
<u>BIND</u>	<u>68</u>
<u>BOR, ORB</u>	<u>69</u>
<u>BTST</u>	<u>69</u>
<u>BUFOUT</u>	<u>69</u>
<u>CATCH</u>	<u>70</u>
<u>CHECKU</u>	<u>70</u>
<u>CLEAR</u>	<u>70</u>
<u>COLOR</u>	<u>71</u>

<u>COND</u>	<u>71</u>
<u>CRLF</u>	<u>73</u>
<u>CURGET</u>	<u>73</u>
<u>CURSET</u>	<u>73</u>
<u>DCLEAR</u>	<u>74</u>
<u>DEC</u>	<u>74</u>
<u>DIRIN</u>	<u>75</u>
<u>DIROUT</u>	<u>75</u>
<u>DISPLAY</u>	<u>75</u>
<u>DLESS?</u>	<u>76</u>
<u>DO</u>	<u>76</u>
<u>ERASE</u>	<u>80</u>
<u>E?</u>	<u>80</u>
<u>FCLEAR</u>	<u>81</u>
<u>FIRST?</u>	<u>81</u>
<u>FONT</u>	<u>81</u>
<u>FSET</u>	<u>82</u>
<u>FSET?</u>	<u>82</u>
<u>FSTACK</u>	<u>82</u>
<u>G?, GRTR?</u>	<u>83</u>
<u>G=?</u>	<u>83</u>
<u>GET</u>	<u>83</u>
<u>GETB</u>	<u>83</u>
<u>GETP</u>	<u>84</u>
<u>GETPT</u>	<u>84</u>
<u>GVAL</u>	<u>84</u>
<u>HLIGHT</u>	<u>85</u>
<u>IFFLAG</u>	<u>85</u>
<u>IGRTR?</u>	<u>85</u>
<u>IN?</u>	<u>85</u>
<u>INC</u>	<u>86</u>
<u>INPUT</u>	<u>86</u>
<u>INTBL?</u>	<u>87</u>
<u>IRESTORE</u>	<u>87</u>
<u>ISAVE</u>	<u>87</u>
<u>ITABLE</u>	<u>88</u>
<u>L?, LESS?</u>	<u>88</u>
<u>L=?</u>	<u>89</u>
<u>LEX</u>	<u>89</u>
<u>LOC</u>	<u>89</u>



<u>LOWCORE-TABLE</u>	<u>90</u>
<u>LOWCORE</u>	<u>90</u>
<u>LSH, SHIFT</u>	<u>90</u>
<u>LTABLE</u>	<u>90</u>
<u>LVAL</u>	<u>91</u>
<u>MAP-CONTENTS</u>	<u>91</u>
<u>MAP-DIRECTIONS</u>	<u>92</u>
<u>MARGIN</u>	<u>93</u>
<u>MENU</u>	<u>93</u>
<u>MOD</u>	<u>93</u>
<u>MOUSE-INFO</u>	<u>94</u>
<u>MOUSE-LIMIT</u>	<u>94</u>
<u>MOVE</u>	<u>95</u>
<u>N=?, N==?</u>	<u>95</u>
<u>NEXT?</u>	<u>95</u>
<u>NEXTP</u>	<u>95</u>
<u>NOT</u>	<u>96</u>
<u>OR</u>	<u>96</u>
<u>ORIGINAL?</u>	<u>96</u>
<u>PICINF</u>	<u>97</u>
<u>PICSET</u>	<u>97</u>
<u>PLTABLE</u>	<u>97</u>
<u>POP</u>	<u>97</u>
<u>PRINT</u>	<u>98</u>
<u>PRINTB</u>	<u>98</u>
<u>PRINTC</u>	<u>98</u>
<u>PRINTD</u>	<u>99</u>
<u>PRINTF</u>	<u>99</u>
<u>PRINTI</u>	<u>99</u>
<u>PRINTN</u>	<u>99</u>
<u>PRINTR</u>	<u>100</u>
<u>PRINTT</u>	<u>100</u>
<u>PRINTU</u>	<u>100</u>
<u>PROG</u>	<u>101</u>
<u>PTABLE</u>	<u>101</u>
<u>PTSIZE</u>	<u>102</u>
<u>PUSH</u>	<u>102</u>
<u>PUT</u>	<u>103</u>
<u>PUTB</u>	<u>103</u>
<u>PUTP</u>	<u>104</u>

<a href="#"><u>QUIT</u></a>	<a href="#"><u>104</u></a>
<a href="#"><u>RANDOM</u></a>	<a href="#"><u>104</u></a>
<a href="#"><u>READ</u></a>	<a href="#"><u>104</u></a>
<a href="#"><u>REMOVE</u></a>	<a href="#"><u>105</u></a>
<a href="#"><u>REPEAT</u></a>	<a href="#"><u>106</u></a>
<a href="#"><u>REST</u></a>	<a href="#"><u>106</u></a>
<a href="#"><u>RESTART</u></a>	<a href="#"><u>107</u></a>
<a href="#"><u>RESTORE</u></a>	<a href="#"><u>107</u></a>
<a href="#"><u>RETURN</u></a>	<a href="#"><u>108</u></a>
<a href="#"><u>RFALSE</u></a>	<a href="#"><u>108</u></a>
<a href="#"><u>RFATAL</u></a>	<a href="#"><u>109</u></a>
<a href="#"><u>RSTACK</u></a>	<a href="#"><u>109</u></a>
<a href="#"><u>RTRUE</u></a>	<a href="#"><u>109</u></a>
<a href="#"><u>SAVE</u></a>	<a href="#"><u>109</u></a>
<a href="#"><u>SCREEN</u></a>	<a href="#"><u>110</u></a>
<a href="#"><u>SCROLL</u></a>	<a href="#"><u>110</u></a>
<a href="#"><u>SET</u></a>	<a href="#"><u>110</u></a>
<a href="#"><u>SETG</u></a>	<a href="#"><u>111</u></a>
<a href="#"><u>SOUND</u></a>	<a href="#"><u>111</u></a>
<a href="#"><u>SPLIT</u></a>	<a href="#"><u>111</u></a>
<a href="#"><u>T?</u></a>	<a href="#"><u>112</u></a>
<a href="#"><u>TABLE</u></a>	<a href="#"><u>112</u></a>
<a href="#"><u>TELL</u></a>	<a href="#"><u>113</u></a>
<a href="#"><u>THROW</u></a>	<a href="#"><u>113</u></a>
<a href="#"><u>USL</u></a>	<a href="#"><u>114</u></a>
<a href="#"><u>VALUE</u></a>	<a href="#"><u>114</u></a>
<a href="#"><u>VERIFY</u></a>	<a href="#"><u>114</u></a>
<a href="#"><u>VERSION?</u></a>	<a href="#"><u>114</u></a>
<a href="#"><u>WINATTR</u></a>	<a href="#"><u>115</u></a>
<a href="#"><u>WINGET</u></a>	<a href="#"><u>115</u></a>
<a href="#"><u>WINPOS</u></a>	<a href="#"><u>115</u></a>
<a href="#"><u>WINPUT</u></a>	<a href="#"><u>115</u></a>
<a href="#"><u>WINSIZE</u></a>	<a href="#"><u>116</u></a>
<a href="#"><u>XPUSH</u></a>	<a href="#"><u>116</u></a>
<a href="#"><u>ZWSTR</u></a>	<a href="#"><u>116</u></a>
<a href="#"><u>Appendix A: Other Z-machine OP-codes</u></a>	<a href="#"><u>117</u></a>
<a href="#"><u>Appendix B – Field-spec for header</u></a>	<a href="#"><u>118</u></a>
<a href="#"><u>Ordinary header</u></a>	<a href="#"><u>118</u></a>
<a href="#"><u>Extended header</u></a>	<a href="#"><u>120</u></a>
<a href="#"><u>Appendix C - Reserved constants, globals &amp; locals</u></a>	<a href="#"><u>120</u></a>



# ZIL Reference Guide

## Introduction

Historically Zork (the mainframe version) was developed in MDL at M.I.T. On an PDP-10 ITS. When Infocom faced the task of moving Zork to 8-bit computers they created a virtual machine that was able to run a subset of MDL (just enough to get a stripped down version of Zork to run, Zork I). This virtual machine is now often called a "Z-Machine", and exists in many versions on many platforms.

The Z-machine runs this subset of commands and reads the game data from a formatted data-structure suited from Interactive Fiction.

On Infocom the developing environment always was in MDL on PDP-10. In this environment they had access to MDL and a library of `FUNCTIONS` designed to help build the data-structure. In the environment there was also `ZILCH` that compiled the code to a format that the Z-machine could understand.

This means that everything that is inside a `ROUTINE` is code that compiles to instructions that the Z-machine understands and everything that is outside the `ROUTINE` is MDL that is used to build the data-structure. There are two classes of commands. And some instructions to `ZILCH`, the compiler.

The full developing environment for Infocom doesn't exist today, even though parts exist in a PDP-10 ITS emulation project. As of today there is a MDL interpreter and some code of `ZILCH`, but primarily the MDL compiler is still missing. Efforts are made to piece together the PDP-10 ITS environment from old tapes and eventually it may succeed.

Luckily there is now another way to write and compile ZIL, ZILF.

The ZILF environment contains a subset of MDL and the Infocom library of `FUNCTIONS` (to build the data-structure and `ROUTINES`). ZILF also can compile all this to a format that then can run in a Z-machine.

This document is divided in basically two parts.

The first part is the things that only work outside a `ROUTINE`. These commands are processed during compilation to build the data-structure. Here you need to pay attention to order and declare things before they are used.

The second part is things that only work inside a `ROUTINE`. These commands are processed by the Z-machine during runtime.

Sources:

*Learning ZIL, Steve E. Meretzky*

*ZIL Course, Marc S. Blank*

## Syntax

Typename	Size	Min-Max	Examples
FIX	32-bit signed integer	-2147483648 to 2147483648	616 *747*

			#2 10110111
CHARACTER	8-bit	0 to 255	!\A
BYTE	8-bit	0 to 255	65
FALSE			<>
<CHTYPE value type> <GVAL value> <LIST values ...> <LVAL value> <VECTOR values ...> <QUOTE value>	#type value ,value (values ...) .value [values ...] 'value		

### ***Regarding TRUE and FALSE***

True and false are handled differently depending on if you are "outside" or "inside" routines.

Outside routines FALSE is its own TYPE which evaluates to an empty list <>.

Inside routines the value 0 is considered FALSE, all other values are considered TRUE.

Example:

`<=? <> 0>`      `--> FALSE "outside", but TRUE "inside"`

## **MDL builtins and ZIL library (use outside ROUTINE)**

The syntax for most of these commands are much like the syntax in MDL.

All these commands are possible to run, test and debug during the interactive mode of ZILF (start ZILF without any options).

Sources:

*The MDL Programming Language, S. W. Galley and Greg Pfister*

*ZIL Language Guide, Jesse McGrew*

### **\* (multiply)**

```
<* numbers ...>
```

MDL builtin

Multiply numbers.

Example:

```
<* 2 3 4> --> 24
```

### **+ (add)**

```
<+ numbers ...>
```

MDL builtin

Add numbers.

Example:

```
<+ 2 3 4> --> 7
```

### **- (subtract)**

```
<- numbers ...>
```

MDL builtin

Subtract first number by subsequent numbers

If only one number is provided, it's subtracted from zero (i.e. negated).

Examples:

```
<- 8 3 4> --> 1  
<- 5>      --> -5
```

### **/ (divide)**

```
</ numbers ...>
```

MDL builtin

Divide first number by subsequent numbers.

Example:

```
<* 20 5 2>      -->  2
```

## 0?

```
<0? value>
```

MDL builtin

Predicate. True if value is 0 otherwise false.

## 1?

```
<1? value>
```

MDL builtin

Predicate. True if value is 1 otherwise false.

## ==?

```
<==? value1 value2>
```

MDL builtin

Predicate. True if value1 and value2 is the same object, otherwise false.

ZILF defines "the same object" more loosely than MDL did:

STRINGS are considered ==? if they contain the same text.

LVALs and GVALs are considered ==? if they refer to the same ATOMs.

Examples:

```
<SET X 1>
<==? .X 1>      -->  True

<SET X (1 2 3)>
<==? .X (1 2 3)>  -->  False

<==? "Hello" "Hello">  -->  True (ZILF not in MDL)
```

## =?

```
<=? value1 value2>
```

MDL builtin

Predicate. True if value1 and value2 is of the same TYPE and structurally equal, otherwise false.

Examples:

```
<SET X 1>
<=? .X 1>          -->  True

<SET X (1 2 3)>
<=? .X (1 2 3)>-->  True
```

## ADD-TELL-TOKENS

```
<ADD-TELL-TOKENS {pattern form} ...>
```

ZIL library

Add a new pattern and form to the current TELL-TOKENS. These can then be used in TELL.

Each pattern starts with either:

- Any single ATOM except \* (asterisk)
- A LIST of ATOMS, which will define them as synonyms

A simple pattern, like CR, consists of a name and nothing else. More often, patterns also define placeholders to match -- and optionally capture -- parameter values when the token is used inside a TELL. The rest of the pattern consists of any number of:

- An asterisk ( \* ), to match and capture any value.
- An ADECL whose left side is an asterisk (like \*:FIX ), to match and capture any value that matches the DECL pattern on the right side.
- A GVAL (like , PRSO or equivalently <GVAL PRSO> ), to match that exact GVAL without capturing it.

Each pattern is followed by a form that will be copied and inserted in place of the TELL when the pattern is matched. Each element of the form must be either:

- An ATOM, FIX, STRING, or FALSE.
- An LVAL or GVAL
- An empty FORM

The form must contain exactly one LVAL for each element of the pattern that captures a value. These LVALS are positional placeholders that will be replaced by the captured values, in order. The specific ATOM referenced by each LVAL is ignored.

Example (zilib 0.9 adds these tokens):

```
<ADD-TELL-TOKENS
  T *          <PRINT-DEF .X>
  A *          <PRINT-INDEF .X>
  CT *         <PRINT-CDEF .X>
  CA *         <PRINT-CINDEF .X>
  NOUN-PHRASE * <PRINT-NOUN-PHRASE .X>
  OBJSPEC *    <PRINT-OBJSPEC .X>
  SYNTAX-LINE * <PRINT-SYNTAX-LINE .X>
  WORD *       <PRINT-WORD .X>
```



```
MATCHING-WORD * * * <PRINT-MATCHING-WORD .X .Y .Z>>
```

## ADD-WORD

```
<ADD-WORD atom-or-string [part-of-speech] [value] [flags]>
```

## ADJ-SYNONYM

```
<ADJ-SYNONYM original synonyms ...>
```

## AGAIN

```
<AGAIN [activation]>
```

MDL builtin

AGAIN means “start doing this again”, where “this” is specified by the activation. If no activation is supplied AGAIN starts evaluating from the last automatically created activation (PROG and REPEAT automatically creates an activation). The evaluation is not redone completely: in particular, no re-binding (of arguments, "AUX" variables, etc.) is done.

Examples:

```
<DEFINE TEST-AUTO-ACT ()
  <PROG ((X 0))
    <SET X <+ .X 1>>
    <PRIN1 .X>
    <COND (<=? .X 5> <RETURN T>)>
    <AGAIN>
  >
>
```

```
<DEFINE TEST-NAMED-ACT-1 ACT ("AUX" (X 0))
  <SET X <+ .X 1>>
  <PRIN1 .X>
  <COND (<=? .X 5> <RETURN T .ACT>)>
  <AGAIN .ACT>
>
```

```
<DEFINE TEST-NAMED-ACT-2 ("NAME" ACT "AUX" (X 0))
  <SET X <+ .X 1>>
  <PRIN1 .X>
  <COND (<=? .X 5> <RETURN T .ACT>)>
  <AGAIN .ACT>
>
```

## ALLTYPES

```
<ALLTYPES>
```

MDL builtin

returns a VECTOR containing the ATOMS which can currently be returned by TYPE or PRIMTYPE.

## AND

<AND expressions...>

MDL builtin

Boolean AND. Requires that all expressions evaluate to true to return true. Exits on the first expression that evaluates to false (rest of expressions are not evaluated).

Because 0 is considered false and all other values are considered true inside a routine AND returns 0 if one expression is false or the value of the last expression if all expressions are true.

Because false is its own TYPE outside a routine AND returns #FALSE if one of the expressions is false or the value of the last expression if all expressions are true.

Example:

```
<AND <=? 1 1> <N=? 1 2>> --> True
<AND <=? 1 2> <SET X 2>> --> X never set to 2 because
                             first predicate evaluates
                             to false
<SET X <AND 1 2 3 0 4>> --> X is set to 4
<SET X <AND 1 2 3 <> 4>> --> X is set to #FALSE
<SET X <AND 1 4 3 2>> --> X is set to 2
```

## AND?

<AND? expressions ...>

MDL builtin

Returns the same result as AND with the difference that all expressions are evaluated.

Examples:

```
<AND? <=? 1 1> <N=? 1 2>>--> True
<AND? <=? 1 2> <SET X 2>>--> X is set to 2 because
                             all expressions are
                             evaluated
```

## ANDB

<ANDB numbers ...>

MDL builtin

Bitwise AND.

Examples:

```
<ANDB 33 96> --> 32
<ANDB 33 96 64> --> 0
```

## APPLICABLE?

```
<APPLICABLE? value>
```

MDL builtin

Predicate. Returns true if TYPE of value is of an applicable TYPE.

Applicable TYPE:s:

```
FIX
FSUBR
FUNCTION
MACRO
OFFSET
SUBR
```

Example:

```
<DEFINE SQR (X) <* .X .X>>
```

```
<APPLICABLE? ,SQR> --> True
```

## APPLY

```
<APPLY applicable args ...>
```

MDL builtin

Call the applicable with args. <APPLY applicable args ...> is equivalent to <applicable args ...>. applicable must be an atom that APPLICABLE? evaluates to true (usually FUNCTION, SUBR, FSUBR & MACRO). APPLY is often used when the applicable to be called is resolved during run-time (dispatch-table).

Examples:

```
<CONSTANT DISPATCH-TBL <VECTOR FUNC1 FUNC2>>
<DEFINE FUNC1 (X) <* .X .X>>
<DEFINE FUNC2 (X) <* .X .X .X>>
    <APPLY ,<NTH ,DISPATCH-TBL 1> 2> --> 4
<APPLY ,<NTH ,DISPATCH-TBL 2> 2> --> 8
```

## APPLYTYPE

```
<APPLYTYPE atom [handler]>
```

## ASCII

```
<ASCII {number | character}>
```

MDL builtin

Converts number to character or character to number.

Examples:

```
<ASCII !\A>      --> 65
<ASCII 65>       --> !\A
```

## ASSIGNED?

```
<ASSIGNED? atom [environment]>
```

MDL builtin

Predicate. Returns true if the atom has an LVAL (local value).

It is possible to supply an environment for ASSIGNED?. See EVAL for more about the environment.

Example:

```
<ASSIGNED? X>  --> False
<SET X 1>
<ASSIGNED? X>  --> True
```

## ASSOCIATIONS

```
<ASSOCIATIONS>
```

## ATOM

```
<ATOM pname>
```

## AVALUE

```
<AVALUE asoc>
```

## BACK

```
<BACK array [count]>
```

MDL builtin

Moves count elements back in array. If count moves past the start of the array an error is raised. Default value for count is 1.

BACK only works on the structures VECTOR or STRING (arrays) and not on a LIST (a LIST is only pointing forward).

Note that the returned array is not a copy but pointing to the same array with another starting element.

Also see LENGTH, NTH, PUT, REST, SUBSTRUC and TOP.

Example:

```
<SETG STRUCT1 [1 2 3 4 5]>      --> STRUCT1 = [1 2 3 4 5]
<SETG STRUCT2 <REST ,STRUCT1 2>> --> STRUCT2 = [3 4 5]
<BACK ,STRUCT2 1>                --> STRUCT2 = [2 3 4 5]
```

## BIND

```
<BIND [activation] (bindings ...) [decl] expressions ...>
```

MDL builtin

BIND defines a program block with its own set of bindings. BIND is similar to PROG and REPEAT but BIND doesn't create a default activation (like PROG and REPEAT) at the start of the block and don't have an automatic AGAIN at the end of the block (like REPEAT). If an activation is needed it must be specified. AGAIN and RETURN without specified activation inside a BIND-block will start over or return from the closest surrounding activation within the current function.

The decl is used to specify the valid TYPE of the variables. In its simplest form decl is formatted like: #DECL ((X) FIX), meaning that X must be of the TYPE FIX. For more information on how to format the decl see *The MDL Programming Language, chap. 14*.

Also see AGAIN, PROG, REPEAT and RETURN for more details how to control program flow.

Example:

```
<BIND ((X 1)) #DECL ((X) FIX)
  <BIND ((X 2)) <PRIN1 .X>> <PRIN1 .X>>
--> "21"

<DEFINE TEST-BIND-AS-REPEAT ()
  <PRINC "START ">
  <BIND ACT ((X 0))
    <SET X <+ .X 1>>
    <PRIN1 .X>
    <COND (<=? .X 3> <RETURN T .ACT>)> ;"--> exit
                                     block"
    <AGAIN .ACT> ;"--> repeat"
  >
  <PRINC " END">
>

<TEST-BIND-AS-REPEAT> --> "START 123 END"
```

## BIT-SYNONYM

```
<BIT-SYNONYM first synonyms ...>
```

## BLOCK

```
<BLOCK (oblist ...)>
```

## BOUND?

```
<BOUND? atom [environment]>
```

MDL builtin

BOUND? is a predicate that returns true if the atom ever had a local value in the environment.

If no environment is supplied, the environment defaults to current scope. See EVAL for more about the environment.

Examples:

```
<SET X 42>
<ASSIGNED? X>  -->  True
<GBOUND? X>   -->  True
<GUNASSIGN X>
<GASSIGNED? X> -->  False
<GBOUND? X>   -->  True
```

## BUZZ

```
<BUZZ atoms ...>
```

## BYTE

```
<BYTE number>
#BYTE number           ;"Alternative syntax (MDL builtin)"
<CHTYPE number BYTE>;"Alternative syntax (MDL builtin)"

ZIL library
```

BYTE changes number of TYPE to #BYTE.

Examples:

```
<BYTE 42>           -->  #BYTE 42
#BYTE 42             -->  #BYTE 42
<CHTYPE 42 BYTE>    -->  #BYTE 42
```

## CHECK-VERSION?

```
<CHECK-VERSION? Version-spec>
```

## CHRSET

```
<CHRSET alphabet-number {string | character |
                           number | byte} ...>
```

## CHTYPE

```
<CHTYPE value type>
#type value           ;"Alternative syntax"

MDL builtin
```

CHTYPE returns a new object that has TYPE type and the same “data part” as value. The PRIMTYPE of value must be the same as the TYPEPRIM of type otherwise an error will be generated.

There is a abbreviated form to change type by typing #type value instead.

Examples:

```

<CHTYPE !\A FIX>
--> 65
#FIX !\A
--> 65
#LIST [1 2 3]
--> ERROR

```

## CLOSE

```
<CLOSE channel>
```

## COMPILATION-FLAG

```
<COMPILATION-FLAG atom-or-string [value]>
```

## COMPILATION-FLAG-DEFAULT

```
<COMPILATION-FLAG-DEFAULT atom-or-string value>
```

## COMPILATION-FLAG-VALUE

```
<COMPILATION-FLAG-VALUE atom-or-string>
```

## COND

```
<COND (condition body ...) ...> **F
```

## CONS

```
<CONS first list>
```

```
MDL builtin
```

CONS (“construct”) adds `first` to the front of `list`, without copying `list`, and returns the resulting LIST. References to `list` are not affected.

Examples:

```

<CONS 1 (2 3)>                -->  (1 2 3)

<SET S1 (!\B !\C)>
<SET S2 <CONS !\A .S1>>
<PUT .S1 2 !\D>
.S2                            -->  (!\A !\B !\D)

```

## CONSTANT

```
<CONSTANT atom value>
```

```
ZIL library
```

CONSTANT defines an atom with value that will never be changed. The atom can be accessed inside a ROUTINE with GVAL (or ,) just like a GLOBAL atom. Defining a CONSTANT instead of a GLOBAL when possible can be vital information the compiler can use for optimization.

MSETG is an alias for CONSTANT.

Example:

```
<CONSTANT MSG-CANT-DO-THAT "You can't do that!">
...
<TELL ,MSG-CANT-DO-THAT CR>
```

## CRLF

```
<CRLF [channel]>
```

MDL builtin

Prints a carriage-return and a line-feed to channel (default for channel is <LVAL OUTCHAN> - the console). CRLF returns true.

Example:

```
<CRLF>      -->  "\n"
```

## DECL-CHECK

```
<DECL-CHECK boolean>
```

## DECL?

```
<DECL? value pattern>
```

## DEFAULT-DEFINITION

```
<DEFAULT-DEFINITION name body ...>
```

ZIL library

This defines a “replaceable” block with the given name.

If neither DELAY-DEFINITION nor REPLACE-DEFINITION was previously called for the given name, then the body is evaluated, and this function returns the result of evaluating the last element of the body.

If the block was replaced (via REPLACE-DEFINITION), the replacement body supplied earlier is used instead.

If the block was delayed (via DELAY-DEFINITION), the body is ignored, and this function returns FALSE.

It is possible to do the same by setting REDEFINE to true. This actually makes it possible to change ALL definitions (it is the last one that becomes the one actually compiled).

Example:

```
<REPLACE-DEFINITION MY-ROUTINE
  <ROUTINE MY-ROUTINE ()
    <TELL "Replaced version of MY-ROUTINE" CR>
  >
```



```

>
<DEFAULT-DEFINITION MY-ROUTINE
  <ROUTINE MY-ROUTINE ()
    <TELL "Original version of MY-ROUTINE" CR>
  >
>
<MY-ROUTINE>          -->  "Replaced version of MY-ROUTINE"

;"Alternative way"
<ROUTINE MY-ROUTINE ()
  <TELL "Original version of MY-ROUTINE" CR>
>
<SET REDEFINE T>
  <ROUTINE MY-ROUTINE ()
    <TELL "Replaced version of MY-ROUTINE" CR>
  >
<SET REDEFINE <>>
<MY-ROUTINE>          -->  "Replaced version of MY-ROUTINE"

```

## DEFINE

```
<DEFINE name [activation] arg-list [decl] expressions ...>
```

MDL builtin

DEFINE assigns the global variable name with a FUNCTION. See FUNCTION for an explanation of activation, arg-list, decl and expressions.

<DEFINE name ...> is equivalent to <SETG name #FUNCTION ...> with the exception that DEFINE protects from overwriting name with a new FUNCTION (this behaviour can be changed by setting REDEFINE to true, instead of false).

Example:

```

<DEFINE MYADD (X1 X2) <+ .X1 .X2>>
<MYADD 4 5>                                -->  9

<DEFINE SQUARE (X) <* .X .X>>
<SQUARE 5>                                -->  25

<DEFINE POWER-TO ACT (X "OPT" (Y 2))
  <COND (<=? .Y 0> <RETURN 1 .ACT>)>
  <REPEAT ((Z 1) (I 0))
    <SET Z <* .Z .X>>
    <SET I <+ .I 1>>
    <COND (<=? .I .Y> <RETURN .Z>)>
  >
>
<POWER-TO 2 3>                             -->  8
<POWER-TO 3 4>                             -->  81

```

## DEFINE-GLOBALS

```
<DEFINE-GLOBALS group-name
  (atom-or-adecl [{BYTE | WORD}] [initializer]) ...>
```

```
ZIL library
```

Defines a set of macros that can be used for global storage in Z-code, similar to global variables.

Each `atom-or-adecl` becomes the name of a new macro which can be called with no arguments (to read the global value) or one argument (to write it). The optional `initializer` sets the initial value, as in `GLOBAL`. `BYTE` or `WORD` can be specified to set the global's size; `WORD` is the default.

ZILF ignores the `group-name`.

See `FUNNY-GLOBALS?` for a more convenient way to bypass the Z-machine's global variable limit. (In fact, ZILF implements `DEFINE-GLOBALS` by turning on `FUNNY-GLOBALS?` and defining a global variable for each macro.)

## DEFINE20

```
<DEFINE20 name [activation] arg-list [decl] expressions ...>
```

```
ZIL library
```

`DEFINE20` is an alias for `DEFINE` except that it isn't affected by MDL-ZIL mode: it always defines a MDL function.

`DEFINE20` (and `SETG20`) are used in "MDL-ZIL"-files, where routines are defined with `DEFINE` instead of `ROUTINE`, global variables are created with `SETG` instead of `GLOBAL`, etc. Presumably that was a way to run the games in MDL during development to avoid recompiling them. `SETG20` and `DEFINE20` are aliases for the MDL versions of `SETG` and `DEFINE`.

## DEFINITIONS

```
<DEFINITIONS package-name>
```

## DEFMAC

```
<DEFMAC name [activation-atom] arg-list [decl] body ...> **F
```

## DEFSTRUCT

```
<DEFSTRUCT
  type-name {base-type | (base-type struct-options ...)}
  (field-name decl field-options ...) ...> **F
```

## DELAY-DEFINITION

```
<DELAY-DEFINITION name>
```

## DIR-SYNONYM

```
<DIR-SYNONYM original synonyms ...>
```

## DIRECTIONS

```
<DIRECTIONS atoms ...>
```

## EMPTY?

```
<EMPTY? structure>
```

MDL builtin

Predicate. Returns true if structure contains no elements, otherwise false.

structure must be an object that STRUCTURED? evaluates to true.

Examples:

```
<EMPTY? [1 2 3]>    -->  False
<EMPTY? []>         -->  True
```

## END-DEFINITIONS

```
<END-DEFINITIONS>
```

## ENDBLOCK

```
<ENDBLOCK>
```

## ENDPACKAGE

```
<ENDPACKAGE>
```

## ENDSECTION

```
<ENDSECTION>
```

## ENTRY

```
<ENTRY atoms ...>
```

## EQVB

```
<EQVB numbers ...>
```

MDL builtin

Bitwise equivalence (inverse of exclusive “or”). Uses 32-bit.

Examples:

```
<XORB 250 245> --> 00000000 00000000 00000000 11111010
                    00000000 00000000 00000000 11110101
                    -----
```

11111111 11111111 11111111 11110000 = -16

## ERROR

<ERROR values ...>

## EVAL

<EVAL value [environment]>

MDL builtin

This evaluates value (usually a FORM created by FORM or QUOTE).

It is possible to supply an environment for EVAL. This tells EVAL from which environment EVAL should take variable bindings. See *The MDL Programming Language, chap. 9.7* for more about the environment.

Examples:

```
<SET F '<+ 1 2>>
.F                               -->  <+ 1 2>
<EVAL .F>                       -->  3

<SET A 0>
<DEFINE WRONG ('B "AUX" (A 1)) <EVAL .B>>
<DEFINE RIGHT ("BIND" E 'B "AUX" (A 1)) <EVAL .B .E>>

<WRONG .A>                      -->  1
<RIGHT .A>                      -->  0
```

## EVALTYPE

<EVALTYPE atom [handler]>

## EXPAND

<EXPAND value>

## FILE-FLAGS

<FILE-FLAGS {CLEAN-STACK? | MDL-ZIL?} ...>

ZIL library

CLEAN-STACK? tells the compiler to generate extra code to remove unneeded values from the stack. Without it, the compiler will generate smaller code in some cases, at the risk of potentially causing stack overflow at runtime.

## FILE-LENGTH

<FILE-LENGTH channel>

## FLOAD

```
<FLOAD filename>
```

## FORM

```
<FORM values ...>
```

MDL builtin

This creates a `FORM` without evaluating it. This is analogous to `LIST` and `VECTOR` but with "`<>`" instead of "`()`" or "`[]`". In many cases it is possible to use `QUOTE` to achieve the same result.

Examples:

```
<FORM + 1 2>          -->  <+ 1 2>

<DEFINE INC-FORM (A)
  <FORM SET .A <FORM + 1 <FORM LVAL .A>>>>

<INC-FORM X>          -->  <SET X <+ 1 .X>
```

## FUNCTION

```
<FUNCTION [activation] arg-list [decl] expressions ...>
#FUNCTION ([activation] arg-list [decl] expressions ...)
```

MDL builtin

This creates a `FUNCTION`. When a `FUNCTION` is called it evaluates all the expressions and returns the result of the last expression.

The `arg-list` is a `LIST` of arguments for the `FUNCTION`. Besides the arguments to the `FUNCTION`, `arg-list` can also contain these tokens (in this order):

"BIND"	Followed by an <code>ATOM</code> that binds the <code>ATOM</code> to the <code>ENVIRONMENT</code> when the <code>FUNCTION</code> was applied. See <code>EVAL</code> for example on this.
Arguments	The required arguments for this <code>FUNCTION</code> . The arguments are bound to local variables inside this <code>FUNCTION</code> .
"OPT"	The optional arguments for this <code>FUNCTION</code> . The arguments are bound to local variables inside this <code>FUNCTION</code> and can be defined with a default value. "OPTIONAL" is an alias for "OPT".
"ARGS"	Followed by an <code>ATOM</code> that is bound a <code>LIST</code> of all remaining arguments, unevaluated. If "ARGS" appears in <code>arg-list</code> , "TUPLE" should not appear.
"TUPLE"	Followed by an <code>ATOM</code> that is bound a <code>TUPLE</code> of all remaining arguments, evaluated. If "TUPLE" appears in <code>arg-list</code> , "ARGS" should not appear. See <code>TUPLE</code> for example on this.
"AUX"	Followed by any number of <code>ATOMs</code> that becomes local variables inside this <code>FUNCTION</code> and can be defined with a default value. "EXTRA" is a alias for "AUX".
"NAME"	Followed by an <code>ATOM</code> that becomes the activation for this <code>FUNCTION</code> . This is equivalent to naming the activation before the <code>arg-list</code> .

"ACT" is an alias for "NAME". See AGAIN for example on this.

Default values for "OPT" and "AUX" are defined by a two-element LIST whose first element is the ATOM and the second element is assigned to.

```
<FUNCTION ("AUX" (X 1) (Y 2)) <+ .X .Y>>
```

Means that the local variables X and Y are initially assigned 1 and 2.

The decl is used to specify the valid TYPE of the variables. In its simplest form decl is formatted like: #DECL ((X) FIX), meaning that X must be of the TYPE FIX. For more information on how to format the decl see *The MDL Programming Language, chap. 14*.

FUNCTION is its own TYPE and it is perfectly legal to, for example, use #FUNCTION instead to create a FUNCTION.

Usually a FUNCTION is assigned to a global variable. This can be done by assigning a global ATOM the FUNCTION with SETG (this is more commonly done with DEFINE).

Examples:

```
<<FUNCTION (X1 X2) <+ .X1 .X2>> 5 4>      --> 9
<SETG SQUARE <FUNCTION (X) <* .X .X>>>
<SQUARE 5>                                --> 25
<SETG POWER-TO <FUNCTION ACT (X "OPT" (Y 2))
  <COND (<=? .Y 0> <RETURN 1 .ACT>)>
  <REPEAT ((Z 1) (I 0))
    <SET Z <* .Z .X>>
    <SET I <+ .I 1>>
    <COND (<=? .I .Y> <RETURN .Z>)>
  >
>>
<POWER-TO 2 3>                            --> 8
<POWER-TO 3 4>                            --> 81
<POWER-TO 3 0>                            --> 1
```

## FUNNY-GLOBALS?

```
<FUNNY-GLOBALS? [boolean]>
```

ZIL library

When enabled, “funny globals” mode lets the game define more than the usual 240 global variables.

If needed, ZILF will move the extra variables into a table (GLOBAL-VARS-TABLE) and generate table instructions to access them (PUT and GET, or in the case of BYTE globals created with DEFINE-GLOBALS, PUTB and GETB).

This translation is mostly transparent to game source code, but it can’t be used for global variables that are ever referenced indirectly by number. ZILF uses a simple heuristic to try to identify those variables and reserve “real” global variable slots for them.

## G=?

```
<G=? value1 value2>
```

MDL builtin

Predicate. True if `value1` is greater or equal than `value2` otherwise false.

## G?

`<G? value1 value2>`

MDL builtin

Predicate. True if `value1` is greater than `value2` otherwise false.

## GASSIGNED?

`<GASSIGNED? Atom>`

MDL builtin

Predicate. Returns true if the `atom` has an GVAL (global value).

Example:

```
<GASSIGNED? X> --> False
<SETG X 1>
<GASSIGNED? X> --> True
```

## GBOUND?

`<GBOUND? atom>`

MDL builtin

GBOUND? Is a predicate that returns true if the `atom` ever had a global value.

Examples:

```
<SETG X 42>
<GASSIGNED? X> --> True
<GBOUND? X> --> True
<GUNASSIGN X>
<GASSIGNED? X> --> False
<GBOUND? X> --> True
```

## GC

`<GC>`

MDL builtin

This causes garbage collection.

In ZILF GC ignores all arguments and always returns true. ZILF relies on the garbage collection in the NET framework and only implements this for compatibility.

Examples:

```
<GC>                -->  T
<GC 0 T 5>          -->  T
```

## GDECL

```
<GDECL (atoms ...) decl ...> **F
```

## GET-DECL

```
<GET-DECL item>
```

## GETB

```
<GETB table index>
```

```
ZIL library
```

Returns BYTE-record (1 byte) stored at index.

TABLE is a ZIL-specific structure that can be used both outside and inside ROUTINES. GETB is equivalent to the Z-code builtin GETB.

Also see PUTB, ZGET, ZPUT and ZREST.

Example:

```
<GETB <TABLE (BYTE) !\A !\B !\C !\D> 2>      -->  !\C
```

## GETPROP

```
<GETPROP item indicator [default-value]>
```

## GLOBAL

```
<GLOBAL atom default-value [decl] [size]>
```

```
ZIL library
```

Declare a global variable atom, that later can be used inside a ROUTINE. The variable is initialized with default-value.

ZILF ignores the decl.

Example:

```
<GLOBAL MYVAR 0>
```

## GROW

```
<GROW structure end beginning>
```

## GUNASSIGN

```
<GUNASSIGN atom>
```



MDL builtin

Unassign global atom.

Example:

```
<SETG X 1>
<GASSIGNED? X>      -->  True
<GUNASSIGN X>
<GASSIGNED? X>      -->  False
```

## GVAL

```
<GVAL atom>
,atom          ;"Alternative syntax"
```

MDL builtin

Get the value of the global atom. More often used in its short form ", atom".

Example:

```
<SETG X 5>

<GVAL X>  -->  5
,X        -->  5
```

## IFFLAG

```
<IFFLAG (condition body ...) ...>
```

ZIL library

Each condition is either:

- A STRING naming a compilation flag, to evaluate the corresponding body if the flag's value is true.
- An ATOM whose PNAME names a compilation flag, to evaluate the corresponding body if the flag's value is true.
- A FORM, to evaluate the FORM after replacing any element ATOMs whose PNAMEs name compilation flags with the flag values, and then evaluate the corresponding body if the result is true.
- Any other value, to evaluate the corresponding body immediately.

As soon as any body is evaluated, the function returns the result. If no body is evaluated, the function returns FALSE.

Note: as a consequence of the evaluation rules above, undefined compilation flags are effectively true .

## ILIST

```
<ILIST count [init]>
```

MDL builtin

ILIST ("implicit" or "iterated") returns a LIST with count items all set to init.

Examples:

```
<ILIST 4 2>                                -->  (2 2 2 2)
<SET A 0>
<ILIST 4 '<SET A <+ .A 1>>>                -->  (1 2 3 4)
```

## IMAGE

```
<IMAGE ch [channel]>
```

## INCLUDE

```
<INCLUDE package-name ...>
```

## INCLUDE-WHEN

```
<INCLUDE-WHEN condition package-name ...>
```

## INDENT-TO

```
<INDENT-TO position [channel]>
```

## INDEX

```
<INDEX offset>
```

## INDICATOR

```
<INDICATOR asoc>
```

## INSERT

```
<INSERT string-or-atom oblist>
```

## INSERT-FILE

```
<INSERT-FILE filename>
```

ZIL library

Insert file with filename at this point. If extension is omitted, ".zil" is assumed.

The filename can have an absolute or relative path. If no path is given, the compiler looks in the current library and the libraries specified to the compiler with the -ip switch.

Note that path is specified like in LINUX (forward slashes etc.) and uppercase/lowercase can be significant, depending on the host system.

Examples:

```
<INSERT-FILE "rooms">                      -->  Include "rooms.zil" from
                                                current directory
```

```
<INSERT-FILE "zillib/parser"> --> Include "parser.zil" from
                                     subdir "zilllib"
```

## ISTRING

```
<ISTRING count [init]>
```

MDL builtin

ISTRING ("implicit" or "iterated") returns a STRING with count items all set to init (character).

Examples:

```
<ISTRING 4 !\A> --> "AAAA"
<SET A 64>
<ISTRING 4 '<ASCII <SET A <+ .A 1>>>> --> "ABCD"
```

## ITABLE

```
<ITABLE [specifier] count [(flags...)] defaults ...>
```

ZIL library

Defines a table of count elements filled with default values: either zeros or, if the default list is specified, the specified list of values repeated until the table is full.

The optional specifier may be the atoms NONE, BYTE, or WORD. BYTE and WORD change the type of the table and also turn on the length marker (element 0 in the table contains the length of the table). This can also be done with the flags (see TABLE about flags).

Examples:

```
<ITABLE 4 0> -->
```

Element 0 WORD	Element 1 WORD	Element 2 WORD	Element 3 WORD
0	0	0	0

```
<ITABLE (BYTE LENGTH) 4 0> -->
```

Element 0 BYTE	Element 1 BYTE	Element 2 BYTE	Element 3 BYTE	Element 4 BYTE
4	0	0	0	0

```
<ITABLE BYTE 4 0> -->
```

Element 0 BYTE	Element 1 BYTE	Element 2 BYTE	Element 3 BYTE	Element 4 BYTE
4	0	0	0	0

TABLE is a ZIL-specific structure that can be used both outside and inside ROUTINES.

## ITEM

<ITEM asoc>

## IVECTOR

<IVECTOR count [init]>

MDL builtin

IVECTOR ("implicit" or "iterated") returns a VECTOR with count items all set to init.

Examples:

```
<IVECTOR 4 2>                                -->  [2 2 2 2]
<SET A 0>
<IVECTOR 4 '<SET A <+ .A 1>>>                -->  [1 2 3 4]
```

## L=?

<L=? value1 value2>

MDL builtin

Predicate. True if value1 is lower or equal than value2 otherwise false.

## L?

<L? value1 value2>

MDL builtin

Predicate. True if value1 is lower than value2 otherwise false.

## LANGUAGE

<LANGUAGE name [escape-char] [change-chrset]>

ZIL library

The language setting changes how text is encoded in two ways: it lets you write language-specific characters in ZIL source code by adding a prefix to ASCII characters, and it changes the Z-machine alphabet to encode them more efficiently.

If change-chrset is false, the Z-machine character set won't be changed, so the language setting will only affect how source code is read.

The escape-char is !\% by default, meaning that language-specific characters may be used in strings or atoms by adding a percent sign prefix (e.g. %s for ß).

The name may be GERMAN, or DEFAULT to stick with classic ZSCII.

GERMAN is defined as follows:

Alphabet 0: abcdefghiklmnoprstuwzäöü.,  
Alphabet 1: ABCDEFGHIKLMNOPRSTUWZj qvxy  
Alphabet 2: 0123456789!?'-:() JÄÖÜß«»  
Special characters: ä(%a), ö(%o), ü(%u), ß(%s), Ä(%A), Ö(%O),  
Ü(%U), «(%<), »(%>)

## LEGAL?

<LEGAL? Value>

## LENGTH

<LENGTH structure>

MDL builtin

Return the number of elements in structure.

structure must be an object that STRUCTURED? evaluates to true.

Note that TABLE is not a structure.

Also see BACK, NTH, PUT, REST, SUBSTRUC and TOP.

Example:

```
<LENGTH <LIST 1 2 3>>          --> 3
```

## LENGTH?

<LENGTH? structure limit>

MDL builtin

LENGTH? is a predicate that returns false if LENGTH of structure is greater than limit, otherwise true (it actually returns LENGTH of structure).

LENGTH? answers the question: "is LENGTH of structure less or equal to limit?"

Examples:

```
<LENGTH? (1 2 3) 1>          --> False  
<LENGTH? (1 2 3) 3>          --> 3  
<NOT <NOT <LENGTH? (1 2 3) 4>>> --> True
```

## LINK

<LINK value str oblist>

## LIST

```
<LIST values ...>  
(values ...) ; "Alternative syntax"
```

MDL builtin

Returns a list of containing values.

A list is a collection of items where each item has a pointer to the next item in the collection. This makes it easy to add and insert items in lists but a list is always forward looking. See more about LIST structure in *The MDL Programming Language, Appendix 1*.

Example:

```
<LIST 1 2 "AB" !\C>      -->  (1 2 "AB" !\C)
(1 2 "AB" !\C)           -->  (1 2 "AB" !\C)
```

## LONG-WORDS?

```
<LONG-WORDS? [boolean]>
```

ZIL library

The `boolean`, which defaults to true if omitted, tells the compiler whether to generate LONG-WORDS-TABLE.

LONG-WORDS-TABLE contains an entry for each vocab word whose length exceeds the maximum word length for the selected Z-machine version (6 Z-characters for V3, or 9 Z-characters for V4+). The table is prefixed by the number of entries, and each entry consists of a word pointer followed by a string giving the printed form of the word.

For example, the table might be defined as equivalent to:

```
<CONSTANT LONG-WORDS-TABLE
  <TABLE 2
    ,W?HEMIDEMIS "hemidemisemiquaver"
    ,W?SUPERCALI "supercalifragilisticexpialidocious">>
```

## LOOKUP

```
<LOOKUP str oblist>
```

## LPARSE

```
<LPARSE text [10] [lookup-oblist]>
```

## LSH

```
<LSH number places>
```

MDL builtin

Bitwise shift. Shift `number` left when `places` is positive and right if it is negative. When right shifting the sign is not preserved (0 is always shifted in).

```
1000 0000 0000 1010      -->  0100 0000 0000 0101
```

Examples:

```
<LSH 4 1>      --> 8
<LSH 4 -2>     --> 1
```

## LTABLE

```
<LTABLE [(flags ...)] values ...>
```

ZIL library

Defines a table containing the specified values and with the LENGTH flag (see TABLE about LENGTH and other flags).

TABLE is a ZIL-specific structure that can be used both outside and inside ROUTINES.

## LVAL

```
<LVAL atom [environment]>
.atom                      ;"Alternative syntax"

MDL builtin
```

Get the value of the local atom. More often used in its short form ".atom".

It is possible to supply an environment for LVAL. See EVAL for more about the environment.

Example:

```
<SET X 5>

<LVAL X>  --> 5
.X        --> 5
```

## M-HPOS

```
<M-HPOS channel>
```

## MAPF

```
<MAPF finalf applicable structs ...>
```

## MAPLEAVE

```
<MAPLEAVE [value]>
```

## MAPR

```
<MAPR finalf applicable structs ...>
```

## MAPRET

```
<MAPRET [value] ...>
```

## MAPSTOP

```
<MAPSTOP [value] ...>
```

## MAX

<MAX numbers ...>

MDL builtin

MAX returns the maximum number among numbers.

Example:

<MAX 2 3 4 1>            -->    4

## MEMBER

<MEMBER item structure>

MDL builtin

MEMBER iterates through structure and returns <REST structure i>, where i is the index of the first element in structure that is =? with item.

MEMBER returns false if the item is not found.

Examples:

```
<MEMBER "BC" "ABCD">            -->    "BCD"
<MEMBER 2 (1 2 3 4)>            -->    (2 3 4)
<MEMBER 0 (1 2 3 4)>            -->    #FALSE <>
```

## MEMQ

<MEMQ item structure>

MDL builtin

MEMQ ("member quick") iterates through structure and returns <REST structure i>, where i is the index of the first element in structure that is ==? with item.

MEMQ returns false if the item is not found.

Examples:

```
<MEMBER "BC" "ABCD">            -->    #FALSE <>
<MEMBER 2 (1 2 3 4)>            -->    (2 3 4)
<MEMBER 0 (1 2 3 4)>            -->    #FALSE <>
```

## MIN

<MIN numbers ...>

MDL builtin

MIN returns the minimum number among numbers.

Example:



```
<MIN 2 3 4 1>      -->  1
```

## MOBLIST

```
<MOBLIST name>
```

## MOD

```
<MOD number1 number2>
```

MDL builtin

MOD divides number1 with number2, which must be non-zero, and returns the remainder.

Examples:

```
<MOD 3 2>      -->  1
<MOD 3256 256> --> 184
```

## MSETG

```
<MSETG atom value>
```

ZIL library

MSETG (Manifest SET Global) is an alias for CONSTANT.

MSETG (CONSTANT) defines an atom with value that will never be changed. The atom can be accessed inside a ROUTINE with GVAL (or ,) just like a GLOBAL atom. Defining a MSETG (CONSTANT) instead of a GLOBAL when possible can be vital information the compiler can use for optimization.

Example:

```
<MSETG MSG-CANT-DO-THAT "You can't do that!">
...
<TELL ,MSG-CANT-DO-THAT CR>
```

## N==?

```
<N==? value1 value2>
```

MDL builtin

Predicate. False if value1 and value2 is the same object, otherwise true. N==? is the opposite to ==?.

Examples:

```
<SET X 1>
<N==? .X 1>      -->  False

<SET X (1 2 3)>
<N==? .X (1 2 3)> -->  True
```

## N=?

```
<N=? value1 value2>
```

MDL builtin

Predicate. False if `value1` and `value2` is of the same TYPE and structurally equal, otherwise true. `N=?` is the opposite to `=?`.

Examples:

```
<SET X 1>
<N=? .X 1>          -->  True

<SET X (1 2 3)>
<N=? .X (1 2 3)>    -->  True
```

## NEW-ADD-WORD

```
<NEW-ADD-WORD atom-or-string [type] [value] [flags]>
```

## NEWTYPE

```
<NEWTYPE name primtype-atom [decl]>
```

## NEXT

```
<NEXT asoc>
```

## NOT

```
<NOT value>
```

MDL builtin

Boolean (logical) "not". NOT returns true if value is false (`#FALSE <>`), otherwise NOT returns false.

Examples:

```
<NOT <>>          -->  T
<NOT T>           -->  #FALSE <>
<NOT <=? 1 2>>    -->  T (Same as <N=? 1 2>)
```

## NTH

```
<NTH structure index>
<index structure>          ;"Alternative syntax"
```

MDL builtin

Returns the element at `index` in `structure`. Valid values for `index` are between 1 and `<LENGTH structure>`.

`structure` must be an object that `STRUCTURED?` evaluates to true.

NTH can also be abbreviated as <index structure>.

Note that TABLE is not a structure.

Also see BACK, LENGTH, PUT, REST, SUBSTRUC and TOP.

Example:

```
<NTH <VECTOR "AB" "CD" "EF"> 2>    --> "CD"
```

## OBJECT

```
<OBJECT name (property values ...) ...>
```

## OBLIST?

```
<OBLIST? Atom>
```

## OFFSET

```
<OFFSET offset structure-decl [value-decl]>
```

## OPEN

```
<OPEN "READ" path>
```

## OR

```
<OR expressions...>
```

MDL builtin

Boolean OR. Requires that one of the expressions evaluates to true to return true. Exits on the first expression that evaluates to true (rest of expressions are not evaluated).

Because false is its own TYPE outside a routine OR returns #FALSE if all expressions are false or the value of the first true expression.

Example:

```
<OR <=? 1 2> <=? 1 1>>          --> True
<OR <=? 1 1> <SET X 2>>         --> X never set to 2 because
                                   first predicate evaluates
                                   to true
<SET X <OR 0 1 2 3>>             --> X is set to 0
<SET X <OR <> 1 2 3>>            --> X is set to 1
```

## OR?

```
<OR? Expressions ...>
```

MDL builtin

Returns the same result as OR with the difference that all expressions are evaluated.

Examples:

```

<OR? <=? 1 2> <=? 1 1>>      -->  True
<OR? <=? 1 1> <SET X 2>>      -->  X is set to 2 because
                                   all expressions are
                                   evaluated

```

## ORB

```
<ORB numbers ...>
```

MDL builtin

Bitwise OR.

Examples:

```

<ORB 33 96>      -->  97
<ORB 33 96 64>   -->  97

```

## ORDER-FLAGS?

```
<ORDER-FLAGS? LAST objects ...>
```

ZIL library

Each of the `objects` is an atom naming a flag, as seen in the `(FLAGS ...)` clause of an `OBJECT` definition.

The only ordering allowed is `LAST`, which causes the named flags to be added to the list of “flags requiring high numbers”, which are assigned the highest flag numbers so they may be distinguished from zero. Flags mentioned in the `(FIND ...)` clause of `SYNTAX` definitions are already added to this list by default.

## ORDER-OBJECTS?

```
<ORDER-OBJECTS? atom>
```

ZIL library

This controls the order in which object numbers are assigned to objects.

Note that there are two ways the compiler can learn about an object: some objects are explicitly “defined” using `ROOM` or `OBJECT`, whereas the existence of others is merely implied when the objects are “mentioned” as part of another object’s definition (in a `LOC` or direction property).

By default, if `ORDER-OBJECTS?` is not used, object numbers are assigned in reverse mention order. That is, the first object defined is given the highest number, and any other objects mentioned in its definition are given the next highest numbers (in order), whether or not those objects are explicitly defined later.

The `atom` is one of the following:

`DEFINED`, to assign numbers to all explicitly defined objects in the order of their definitions (starting at 1), then to all other mentioned objects in the order of their mentions.

ROOMS-FIRST, the same as DEFINED except that numbers are assigned to rooms before non-rooms, so room numbers can be packed into a byte array (assuming there are less than 256 of them).

ROOMS-LAST, the same as DEFINED except that numbers are assigned to non-rooms before rooms.

ROOMS-AND-LGS-FIRST, the same as ROOMS-FIRST except that numbers are assigned to rooms and local globals before the remaining objects.

For the purpose of object ordering, “rooms” include all objects defined with ROOM (instead of OBJECT) as well as all objects whose initial LOC is an object named ROOMS. “Local globals” includes all objects whose initial LOC is an object named LOCAL-GLOBALS.

## ORDER-TREE?

```
<ORDER-TREE? atom>
```

```
ZIL library
```

This controls the initial layout of the Z-machine object tree.

The object tree is defined by three fields on each object, named in the Z-Machine Standards Document as “parent”, “child”, and “sibling”, which are read by the ZIL functions LOC, FIRST?, and NEXT?. Each object’s parent field is specified by the (LOC ...) clause in the object definition, but the compiler has discretion to set the child and sibling fields as long as the tree remains well-formed.

The atom must be:

REVERSE-DEFINED, to force objects to be linked in the reverse order of their definitions. That is, the child of an object X is the last object in the source code whose definition contains (LOC X); the sibling of that child is the next to last object in the source code that contains (LOC X); and so on.

By default, if ORDER-TREE? is not used, the order is the same as REVERSE-DEFINED except for the first defined child, which remains the first object linked. That is, the child of an object X is the first object in the source code whose definition contains (LOC X); the sibling of that child is the last object that contains (LOC X); the sibling of that child in turn is the next to last object that contains (LOC X); and so on.

## PACKAGE

```
<PACKAGE package-name>
```

## PARSE

```
<PARSE text [10] [lookup-oblist]>
```

## PLTABLE

```
<PLTABLE [flags ...] values ...>
```

```
ZIL library
```

Defines a table containing the specified values and with the PURE and LENGTH flag (see TABLE about LENGTH, PURE and other flags).

TABLE is ZIL-specific structure that can be used both outside and inside ROUTINES.

## PNAME

```
<PNAME atom>
```

## PREP-SYNONYM

```
<PREP-SYNONYM original synonyms ...>
```

## PRIMTYPE

```
<PRIMTYPE value>
```

MDL builtin

evaluates to the primitive type of value. The primitive types are ATOM, FIX, LIST, STRING, TABLE and VECTOR.

Examples:

```
<PRIMTYPE !\A>
--> FIX
<PRIMTYPE <+1 2>>
--> FIX
<PRIMTYPE "ABC">
--> STRING
```

## PRIN1

```
<PRIN1 value [channel]>
```

MDL builtin

Prints the evaluated representation of value to channel (default for channel is <LVAL OUTCHAN> - the console). PRIN1 also returns the evaluated representation of value.

Examples:

```
<PRIN1 !\A>                --> !\A
<PRIN1 42>                  --> 42
<PRIN1 "Hello, world!">     --> "Hello, world!"
<PRIN1 (1 2 3)>              --> (1 2 3)
<PRIN1 <+ 1 2>>             --> 3
```

## PRINC

```
<PRINC value [channel]>
```

MDL builtin

PRINC is jut like PRIN1, except for STRING and CHARACTER where surrounding dubbel quote

(") and initial !\ is suppressed. PRINC returns the evaluated representation of value.

Examples:

```
<PRINC !\A>          -->  A
<PRINC 42>           -->  42
<PRINC "Hello, world!"> -->  Hello, world!
<PRINC (1 2 3)>       -->  (1 2 3)
<PRINC <+ 1 2>>      -->  3
```

## PRINT

```
<PRINT value [channel]>
```

MDL builtin

PRINT is just like PRINC, except that it first prints a CRLF, then the evaluated representation of value and lastly a space. PRINT returns the evaluated representation of value.

Examples:

```
<PRINT !\A>          -->  \n!\A<space>
<PRINT 42>           -->  \n42<space>
<PRINT "Hello, world!"> -->  \n"Hello, world!"<space>
<PRINT (1 2 3)>       -->  \n(1 2 3)<space>
<PRINT <+ 1 2>>      -->  \n3<space>
```

## PRINT-MANY

```
<PRINT-MANY channel printer items ...>
```

## PRINTTYPE

```
<PRINTTYPE atom [handler]>
```

## PROG

```
<PROG [activation] (bindings ...) [decl] expressions ...>
```

MDL builtin

PROG defines a program block with its own set of bindings. PROG is similar to BIND and REPEAT but unlike BIND it creates a default activation (like REPEAT) at the start of the block and doesn't have an automatic AGAIN at the end of the block (like REPEAT). It is possible to name an atom to the activation but it is not necessary. AGAIN and RETURN inside a PROG-block will start the block over or return from the block.

The decl is used to specify the valid TYPE of the variables. In its simplest form decl is formatted like: #DECL ((X) FIX), meaning that X must be of the TYPE FIX. For more information on how to format the decl see *The MDL Programming Language, chap. 14*.

Also see AGAIN, BIND, REPEAT and RETURN for more details how to control program flow.

Example:

```
<PROG ((X 1)) #DECL ((X) FIX)
```

```

        <PROG ((X 2)) <PRIN1 .X>> <PRIN1 .X>>
--> "21"

<DEFINE TEST-PROG-AS-REPEAT ()
  <PRINC "START ">
  <PROG ((X 0))
    <SET X <+ .X 1>>
    <PRIN1 .X>
    <COND (<=? .X 3> <RETURN>)> ;"--> exit block"
    <AGAIN> ;"--> repeat"
  >
  <PRINC " END">
>

<TEST-PROG-AS-REPEAT> --> "START 123 END"

```

## PROPDEF

```
<PROPDEF atom default-value spec ...> **F
```

## PTABLE

```
<PTABLE [(flags ...)] values ...>
```

ZIL library

Defines a table containing the specified values and with the PURE flag (see TABLE about PURE and other flags).

TABLE is a ZIL-specific structure that can be used both outside and inside ROUTINES.

## PUT

```
<PUT structure index new-value>
```

MDL builtin

Sets the element at index in structure to new-value. Valid values for index are between 1 and <LENGTH structure>.

structure must be an object that STRUCTURED? evaluates to true.

Note that TABLE is not a structure.

Also see BACK, LENGTH, NTH, REST, SUBSTRUC and TOP.

Example:

```

<SETG STRUCT (1 2 3 4)>
<PUT ,STRUCT 2 5> --> STRUCT = (1 5 3 4)

```

## PUT-DECL

```
<PUT-DECL item pattern>
```



## PUTB

<PUTB table index new-value>

ZIL library

Put a byte new-value in the table at byte position index. Actual address is table-address+index.

TABLE is a ZIL-specific structure that can be used both outside and inside ROUTINES. PUTB is equivalent to the Z-code builtin PUTB.

Also see GETB, ZGET, ZPUT and ZREST.

Example:

```
<PUTB ,MYTABLE 1 !\A>          -->  Stores character A at
                                   position 1 in MYTABLE
```

## PUTPROP

<PUTPROP item indicator [value]>

## PUTREST

<PUTREST list new-rest>

## QUIT

<QUIT [exit-code]>

## QUOTE

<QUOTE value>

'value

MDL builtin

;"Alternative syntax"

QUOTE returns value unevaluated.

Examples:

```
<SET F <QUOTE <+ 1 2>>  -->  Or <SET F '<+ 1 2>>
.F          -->  <+ 1 2>
<EVAL .F> -->  3
```

## READSTRING

<READSTRING dest channel [max-length-or-stop-chars]>

## REMOVE

<REMOVE {atom | pname oblist}>

## REENTRY

```
<REENTRY atoms ...>
```

## REPEAT

```
<REPEAT [activation] (bindings ...) [decl] expressions ...>
```

MDL builtin

REPEAT defines a program block with its own set of bindings. REPEAT is similar to BIND and PROG but unlike BIND it creates a default activation (like PROG) at the start of the block but unlike PROG it also has an automatic AGAIN at the end of the block. It is possible to name an atom to the activation but it is not necessary. A REPEAT-block repeatedly executes expressions until it encounters a RETURN statement that will exit the block.

The decl is used to specify the valid TYPE of the variables. In its simplest form decl is formatted like: #DECL ((X) FIX), meaning that X must be of the TYPE FIX. For more information on how to format the decl see *The MDL Programming Language, chap. 14*.

Also see AGAIN, BIND, PROG and RETURN for more details how to control program flow.

Example:

```
<REPEAT ((X 1)) #DECL ((X) FIX)
  <REPEAT ((X 2)) <PRIN1 .X> <RETURN>>
  <PRIN1 .X> <RETURN>>
--> "21"

<DEFINE TEST-REPEAT ()
  <PRINC "START ">
  <REPEAT ((X 0))
    <SET X <+ .X 1>>
    <PRIN1 .X>
    <COND (<=? .X 3> <RETURN>)> ;"--> exit block"
  >
  <PRINC " END">
>

<TEST-REPEAT> --> "START 123 END"
```

## REPLACE-DEFINITION

```
<REPLACE-DEFINITION name body ...>
```

ZIL library

This tells the compiler this block of code defined by name should replace a later DEFAULT-DEFINITION block of code with the same name.

This is usually used when there is a library that is inserted (like "parser.zil") there some definitions are possible to override.

Note that the REPLACE-DEFINITION is required to appear before the

DEFAULT-DEFINITION.

It is possible to do the same by setting REDEFINE to true. This actually makes it possible to change ALL definitions (it is the last one that becomes the one actually compiled).

See DEFAULT-DEFINITION for examples.

## REST

```
<REST structure [count]>
```

MDL native

Return structure without its first count elements (count is default 1). Note that this is not a copy of the structure, it is pointing to the same structure with another starting element.

structure must be an object that STRUCTURED? evaluates to true.

Note that TABLE is not a structure.

Also see BACK, LENGTH, NTH, PUT, SUBSTRUC and TOP.

Example:

```
<SETG STRUCT1 [1 2 3 4]>          -->  STRUCT1 = [1 2 3 4]
<SETG STRUCT2 <REST ,STRUCT1>>    -->  STRUCT2 = [2 3 4]
<PUT ,STRUCT2 1 5>                 -->  STRUCT1 = [1 5 3 4],
                                     STRUCT2 = [5 3 4]
```

## RETURN

```
<RETURN [value] [activation]>
```

MDL builtin

This returns value from program-block defined by activation. True is returned if no value is specified. If activation is not specified RETURN will exit the current defined program-block where an automatic activation was created (PROG and REPEAT creates automatic activations, BIND does not).

In practice RETURN exits current program-block and returns value to outer program-block defined by BIND (needs activation), PROG or REPEAT.

See AGAIN, BIND, PROG and REPEAT for more examples of using RETURN and details how to control program flow.

Examples:

```
<PROG () <RETURN>>                -->  T
<PROG ACT ()
  <PROG () <RETURN 42 .ACT>>
  <RETURN 43>> ; "Never reached"    -->  42
```

## ROOM

```
<ROOM name (property value ...) ...>
```

## ROOT

<ROOT>

## ROUTINE

<ROUTINE name [activation-atom] arg-list body ...> \*\*F

## ROUTINE-FLAGS

<ROUTINE-FLAGS flags ...>

## SET

<SET atom value [environment]>

MDL builtin

Assign value to the local atom.

It is possible to supply an environment for SET. See EVAL for more about the environment.

Example:

```
<PROG (X) <SET X 5> <RETURN .X>> --> 5
```

## SET-DEFSTRUCT-FILE-DEFAULTS

<SET-DEFSTRUCT-FILE-DEFAULTS args ...> \*\*F

## SETG

<SETG atom value>

MDL builtin

Assign value to the global atom. If an atom already is assigned a value, it is changed.

Example:

```
<SETG MYVAR 42>--> Store 42 in global atom MYVAR
```

## SETG20

<SETG20 atom value>

ZIL library

Assign value to the global atom. If an atom already is assigned a value, it is changed.

SETG20 is an alias for SETG.

Example:

```
<SETG20 MYVAR 42> --> Store 42 in global atom MYVAR
```

## **SORT**

```
<SORT predicate vector [record-size] [key-offset]
  [vector [record-size] ...]>
```

## **SPNAME**

```
<SPNAME atom>
```

## **STRING**

```
<STRING values ...>
```

MDL builtin

STRING returns a concatenated string of all values. values can be character or string.

A string is a block of contiguous bytes where each byte holds a character. See more about STRING structure in *The MDL Programming Language, Appendix 1*.

Example:

```
<STRING !\A <ASCII 66> "CD">      -->  "ABCD"
```

## **STRUCTURED?**

```
<STRUCTURED? value>
```

MDL builtin

Predicate. Returns true if value is of a structured TYPE. The structured TYPE:s are:

```
CHANNEL
DECL
FALSE
FORM
FUNCTION
LIST
MACRO
OBLIST
SEGMENT
SPLICE
STRING
VECTOR
```

Examples:

```
<STRUCTURED? <LIST 1 2 3>>      -->  True
<STRUCTURED? <TABLE 1 2 3>>     -->  False
```

## **SUBSTRUC**

```
<SUBSTRUC structure-from [rest] [amount] [structure-to]>
```

MDL builtin

Copies an amount number of elements, starting at rest, from structure-from. The result is copied into structure-to, if supplied, otherwise a new structure is returned.

Default value for rest is 0 and default value for amount is LENGTH - rest (in other words, copies from rest to end of structure-from).

structure-from must be of PRIMTYPE LIST, VECTOR or STRING and structure-to must be of the same PRIMTYPE as struture-from and have enough room for the SUBSTRUC to fit.

Also see BACK, LENGTH, NTH, PUT, REST and TOP.

Examples:

```
<SUBSTRUC "ABCD" 1 2>          -->  "BC"

<SETG STR1 "EEEEEE">
<SUBSTRUC "ABCD" 1 2 ,STR1>    -->  STR1 = "BCEEEEEEE"
```

## SYNONYM

```
<SYNONYM original synonyms ...>
```

## SYNTAX

```
<SYNTAX verb [prep1] [OBJECT] [(FIND flag-name)]
  [(search-flags ...)] [prep2] [OBJECT]
  [(FIND flag-name)] [(search-flags ...)]
    = action-routine-name [preaction-routine-name]
    [action-name]>
```

## TABLE

```
<TABLE [(flags ...)] values ...>
```

ZIL library

Defines a table containing the specified values.

These flags control the format of the table:

WORD causes the elements to be 2-byte words. This is the default.

BYTE causes the elements to be single bytes.

LEXV causes the elements to be 4-byte records. If default values are given to ITABLE with this flag, they will be split into groups of three: the first compiled as a word, the next two compiled as bytes. The table is also prefixed with a byte indicating the number of records, followed by a zero byte

STRING causes the elements to be single bytes and also changes the initializer format. This flag may not be used with ITABLE. When this flag is given, any values given as strings will be compiled as a series of individual ASCII characters, rather than as string addresses.

These flags alter the table without changing its basic format:

LENGTH causes a length marker to be written at the beginning of the table, indicating the number of elements that follow. The length marker is a byte if BYTE or STRING are also given; otherwise the length marker is a WORD. This flag is ignored if LEXV is given

PURE causes the table to be compiled into static memory (ROM).

The flags LENGTH and PURE are implied in LTABLE, PTABLE or PLTABLE.

Examples:

<TABLE 1 2 3 4> -->

Element 0 WORD	Element 1 WORD	Element 2 WORD	Element 3 WORD
1	2	3	4

<TABLE (BYTE LENGTH) 1 2 3 4> -->

Element 0 BYTE	Element 1 BYTE	Element 2 BYTE	Element 3 BYTE	Element 4 BYTE
4	1	2	3	4

TABLE is a ZIL-specific structure that can be used both outside and inside ROUTINES.

## TELL-TOKENS

<TELL-TOKENS {pattern form} ...>

ZIL library

Replace current TELL-TOKENS with the specified list of pattern and form. These can then be used in TELL.

Example (from Infocom's Trinity):

```
<TELL-TOKENS
(CR CRLF)      <CRLF>
(N NUM) *      <PRINTN .X>
(C CHAR CHR) * <PRINTC .X>
(D DESC) *     <PRINTD .X>
(A AN) *       <PRINTA .X>
THE *          <THE-PRINT .X>
CTHE *         <CTHE-PRINT .X>
THEO           <THE-PRINT>
CTHEO          <CTHE-PRINT>
CTHEI          <CTHEI-PRINT>
THEI           <THEI-PRINT>>
```

## TOP

<TOP array>

MDL builtin

Returns array with all elements put back in `array`.

`TOP` only works on the structures `VECTOR` or `STRING` (arrays) and not on a `LIST` (a `LIST` is only pointing forward).

Note that the returned `array` is not a copy but pointing to the same array with another starting element.

Also see `BACK`, `NTH`, `PUT`, `REST` and `SUBSTRUC`.

Example:

```
<SETG STRUCT1 [1 2 3 4 5]>          -->  STRUCT1 = [1 2 3 4 5]
<SETG STRUCT2 <REST ,STRUCT1 2>>    -->  STRUCT2 = [3 4 5]
<TOP ,STRUCT2>                      -->  STRUCT2 = [1 2 3 4 5]
```

## TUPLE

<TUPLE values ...>

MDL builtin

`TUPLE` is just like a `VECTOR` with the only difference that a `TUPLE` should live on the control stack. The advantage of a `TUPLE` over a `VECTOR` is that a `TUPLE` don't need to be garbage collected, the disadvantage is that a `TUPLE` only lives during the execution of the function where it was declared. It is only valid to declare a `TUPLE` in the "AUX" or "OPTIONAL" part of a functions definition or as a "TUPLE" in a functions definition.

The above is not entirely true for `ZILF`. In `ZILF`, `TUPLE` is treated as an alias to `VECTOR`.

A `TUPLE` defined in the "AUX" or "OPTIONAL" is just like a `VECTOR`. A "TUPLE" definition makes it possible to have a variable number of arguments to a function.

Examples:

```
<DEFINE MY+ ("TUPLE" T)
<REPEAT ((M 0))
  <COND (<EMPTY? .T> <RETURN .M>)>
  <SET M <+ .M <1 .T>>>
  <SET T <REST .T>>
>
>

<MY+ 1 2 3>          -->  6
<MY+ 4 5>            -->  9

<TYPE <TUPLE 1 2 3>>-->  VECTOR (in ZILF!)
                        TUPLE  (in MDL)
```

## TYPE

<TYPE value>



MDL builtin

evaluates to the type of value. Also see ALLTYPES.

Examples:

```
<TYPE !\A>
--> CHARACTER
<TYPE <+1 2>>
--> FIX
<TYPE #BYTE 42>
--> BYTE
```

## TYPE?

```
<TYPE? value type-1 ... type-N>
```

MDL builtin

Evaluates to type-i only if <=? type-i > is true. It is faster and gives more information than ORing tests for each TYPE. If the test fails for all type-i's, TYPE? returns #FALSE ().

Examples:

```
<TYPE? !\A CHARACTER FIX>
--> CHARACTER
<TYPE? <+1 2> CHARACTER FIX>
--> FIX
<TYPE? #BYTE 42 CHARACTER FIX>
--> #FALSE ()
```

## TYPEPRIM

```
<TYPEPRIM type>
```

MDL builtin

evaluates to the primitive type of type. The primitive types are ATOM, FIX, LIST, STRING, TABLE and VECTOR.

Examples:

```
<TYPEPRIM CHARACTER>
--> FIX
<TYPEPRIM FORM>
--> LIST
<TYPEPRIM BYTE>
--> FIX
```

## UNASSIGN

```
<UNASSIGN atom [environment]>
```

MDL builtin

Unassign global atom.

It is possible to supply an environment for ASSIGNED?. See EVAL for more about the environment.

Example:

```
<SET X 1>
<ASSIGNED? X>      -->  True
<UNASSIGN X>
<ASSIGNED? X>      -->  False
```

## UNPARSE

```
<UNPARSE value>
```

## USE

```
<USE package-name ...>
```

## USE-WHEN

```
<USE-WHEN condition package-name ...>
```

## VALID-TYPE?

```
<VALID-TYPE? Atom>
```

## VALUE

```
<VALUE atom [environment]>
```

MDL builtin

This returns the value of an atom. If the atom has an LVAL then the LVAL is returned, otherwise the GVAL of the atom is returned.

It is possible to supply an environment for VALUE. See EVAL for more about the environment.

Example:

```
<SETG X 3>
<SET X 4>
<VALUE X>          ;"--> 4"
<UNASSIGN X>
<VALUE X>          ;"--> 3"
```

## VECTOR

```
<VECTOR values ...>
[values ...]          ;"Alternative syntax"
```

MDL builtin

This returns a vector of containing values.

A vector is a collection of items that occupies a continuous block of memory. This makes it easy to traverse a vector both forward and backward but costly to add or insert items in the vector. See more about VECTOR structure in *The MDL Programming Language, Appendix 1*.

Note that in MDL there is another type of vector, UVECTOR (uniform vector). In an UVECTOR every item is of the same TYPE which makes an UVECTOR more space efficient. ZILF does not support UVECTOR but treats short form definitions of UVECTOR as an ordinary VECTOR (! [1 2 3!] --> [1 2 3]).

Examples:

```
<VECTOR 1 2 "AB" !\C>      -->  [1 2 "AB" !\C]
[1 2 "AB" !\C]              -->  [1 2 "AB" !\C]

<TYPE ![1 2 3!]>           -->  VECTOR (in ZILF)
                               UVECTOR (in MDL)
```

## VERB-SYNONYM

```
<VERB-SYNONYM original synonyms ...>
```

## VERSION

```
<VERSION {ZIP | EZIP | XZIP | YZIP | number} [TIME]>
```

```
ZIL library
```

This tell the compiler which Z-machine version that this program is targeting.

Version	Description
3 or ZIP	Version 3 (file extension *.z3). Almost all classical Infocom games are in this version. You are limited to 255 objects (rooms+items) and the game can't be bigger than 128K.
4 or EZIP	Version 4 (file extension *.z4). Infocom's "plus" games – AMFV, Bureaucracy, Nord and Bert... and Trinity. This format supports 65535 objects and a game size up to 256K.
5 or XZIP	Version 5 (file extension *.z5). Infocom's Beyond Zork, Border Zone, Sherlock and the Solid Gold versions of older games. This version adds things like UNDO, COLOR and timed input. This format supports 65535 objects and a game size up to 256K.
6 or YZIP	Version 6 (file extension *.z6). Infocom's Arthur, Journey, Shohun and Zork Zero. This version primarily adds graphics. This version supports game size up to 512K.
7	Version 7 (file extension *.z7). Post Infocom version. This version supports game size up to 512K. Rarely used version that is superseded by version 8.

8	Version 8 (file extension *.z8). Post Infocom version. This version supports game size up to 512K.
---	--

In version ZIP the status line is drawn by the interpreter and the argument TIME specifies that the status line should display hh:mm instead of score and moves. Global variable 2, usually SCORE, holds the hour-part and global variable 3, usually MOVES, hold the minute-part.

Examples:

```
<VERSION XZIP>          ;"Target Z-machine version 5"
<VERSION 8>              ;"Target Z-machine version 8"
<VERSION ZIP TIME>      ;"Target Z-machine version 3 with hh:mm"
<ROUTINE GO ()
    <SETG SCORE 13>;"Game starting 13:30"
    <SETG MOVES 30>
>
```

## VERSION?

```
<VERSION? (version-spec body ...) ...>
```

ZIL library

VERSION? Tell the compiler to use different code-blocks depending on the setting of VERSION.

The version-spec can be:

```
3      ZIP      4      EZIP
5      XZIP
6      YZIP
7
8
      ELSE/T
```

Example:

```
<VERSION?
    (ZIP <ROUTINE RTN-ZIP () ...>)
    (XZIP <ROUTINE RTN-XZIP () ...>)
    (ELSE <ROUTINE RTN-OTHER () ...>)
>
```

## VOC

```
<VOC string [part-of-speech]>
```

## XORB

```
<XORB numbers ...>
```

MDL builtin

Bitwise exclusive "or".

Examples:

```
<XORB 250 245> --> 11111010 XOR 11110101 = 00001111 (15)
```

## ZGET

```
<ZGET table index>
```

```
ZIL library
```

Returns WORD-record (2 bytes) stored at index.

TABLE is a ZIL-specific structure that can be used both outside and inside ROUTINES. ZGET is equivalent to the Z-code builtin GET.

Also see GETB, PUTB, ZPUT and ZREST.

Example:

```
<ZGET <TABLE 0 1 2 3> 2>          --> 2
```

## ZIP-OPTIONS

```
<ZIP-OPTIONS {COLOR | MOUSE | UNDO | DISPLAY | SOUND  
              | MENU} ...>
```

## ZPUT

```
<ZPUT table index new-value>
```

```
ZIL library
```

Put a 16-bit WORD new-value in the table at word position index. Actual address is table-address+index\*2.

TABLE is a ZIL-specific structure that can be used both outside and inside ROUTINES. ZPUT is equivalent to the Z-code builtin PUT.

Also see GETB, PUTB, ZGET and ZREST.

Examples:

```
<ZPUT ,MYTABLE 1 123>          --> Stores 123 at position 1  
                                in MYTABLE
```

## ZREST

```
<ZREST table bytes>
```

```
ZIL library
```

Return table without its first bytes. Note that this is not a copy of the table, it is pointing to the same table with another starting address.

TABLE is a ZIL-specific structure that can be used both outside and inside ROUTINES. ZREST is equivalent to the Z-code builtin REST.

Also see GETB, PUTB, ZGET and ZPUT.

Example:

<SETG TBL1 <TABLE 1 2 3 4>>	--> TBL1 = [1 2 3 4]
<SETG TBL2 <ZREST ,TBL1 2>>	--> TBL2 = [2 3 4] Move 2 because WORD-table!
<ZPUT ,TBL2 0 5>	--> TBL1 = [1 5 3 4], TBL2 = [5 3 4]

## ZSTART

<ZSTART atom>

ZIL library

Default starting ROUTINE for a compiled ZIL program is the ROUTINE GO. ZSTART can move to ZIL entry point to another ROUTINE.

Example:

<ZSTART MAIN> --> Starts with ROUTINE MAIN instead of GO

## **Z-code builtins (use inside ROUTINE)**

Sources:

*The Z-Machine Standards Document, Graham Nelson*

*The Inform Designer's Manual, Graham Nelson*

*ZIL Language Guide, Jesse McGrew*

### **\*, MUL**

```
<* numbers ...>
<MUL numbers ...>          ;"Alternative syntax"
```

#### **Zapf syntax**

MUL

#### **Inform syntax**

mul

Multiply numbers.

Example:

```
<* 2 3 4> --> 24
```

### **+, ADD**

```
<+ numbers ...>
<ADD numbers ...>          ;"Alternative syntax"
```

#### **Zapf syntax**

ADD

#### **Inform syntax**

add

All versions

Add numbers.

Example:

```
<+ 2 3 4> --> 7
```

### **-, SUB**

```
<- numbers ...>
<SUB numbers ...>          ;"Alternative syntax"
<BACK number1 number2>     ;"Alternative syntax"
```

#### **Zapf syntax**

SUB

#### **Inform syntax**

sub

All versions

Subtract first number by subsequent numbers.

Note that it is possible to use BACK as an alias for SUB.

Example:

```
<- 8 3 4>      --> 1
<BACK 2>       --> 1      (Defaults to 1)
<BACK 1 2>     --> -1
```

## /, DIV

```
</ numbers ...>
<DIV numbers ...>      ;"Alternative syntax"
```

<b>Zapf syntax</b>	<b>Inform syntax</b>
DIV	div

All versions

Divide first number by subsequent numbers.

Example:

```
<* 20 5 2>      --> 2
```

## 0?, ZERO?

```
<0? value>
<ZERO? Value>      ;"Alternative syntax"
```

<b>Zapf syntax</b>	<b>Inform syntax</b>
ZERO?	Jz

All versions

Predicate. True if value is 0 otherwise false.

Example:

```
<0? <- 1 1>>      --> TRUE
```

## 1?

```
<1? value>
```

Predicate. True if value is 1 otherwise false.

Example:

```
<1? <- 2 1>>      --> TRUE
```

## =?, ==?, EQUAL?

```
<=? value1 value2...valueN>
<==? value1 value2...valueN>      ;Alternative syntax"
<EQUAL? value1 value2...valueN>    ;Alternative syntax"
```

<b>Zapf syntax</b>	<b>Inform syntax</b>
--------------------	----------------------



EQUAL?

Je

All versions

**Predicate.** True if value1 is equal to any of the values value2 to valueN.

**Examples:**

```
<=? 1 1>          -->  TRUE
<=? 1 2>          -->  FALSE
<=? 1 2 1>        -->  TRUE
```

## AGAIN

<AGAIN [activation]>

AGAIN means "start doing this again", where "this" is activation. If no activation is supplied the most recent is used. In practice AGAIN is used to restart a program block (BIND, DO, PROG, REPEAT or ROUTINE) again from the top. Note that arguments and variables for a ROUTINE are reinitialized (to starting value, if supplied) otherwise they keep values between iterations. BIND, DO, PROG and REPEAT don't reinitialize variables.

Also see BIND, DO, PROG, REPEAT and RETURN for more details how to control program flow.

**Examples:**

```
<ROUTINE TEST-AGAIN-1 ("AUX" X)
  <SET X <+ .X 1>>
  <TELL N .X " ">
  <COND (<=? .X 5> <RETURN>)>
  <AGAIN>      ;"Start routine again, X keeps value"
>
<TEST-AGAIN-1> -->  "1 2 3 4 5"

<ROUTINE TEST-AGAIN-2 ("AUX" (X 0))
  <SET X <+ .X 1>>
  <TELL N .X " ">
  <COND (<=? .X 5> <RETURN>)> ;"Never reached"
  <AGAIN>      ;"Start routine again, X reinitialize to 0"
>
<TEST-AGAIN-2> -->  "1 1 1 1 1 ..."

<ROUTINE TEST-AGAIN-3 ()
  <BIND ACT1 ((X 0))
  <SET X <+ .X 1>>
  <TELL N .X " ">
  <COND (<=? .X 5> <RETURN>)>
  <AGAIN .ACT1> ;"Start block again from ACT1,"
> ;"X keeps value"
<TEST-AGAIN-3> -->  "1 2 3 4 5"

<ROUTINE TEST-AGAIN-4 ()
  <PROG ((X 0)) ;"PROG generates default activation"
  <SET X <+ .X 1>>
```

```

        <TELL N .X " ">
        <COND (<=? .X 5> <RETURN>)>
    <AGAIN>          ;"Start block again from PROG,"
>                  ;"X keeps value"
<TEST-AGAIN-4> --> "1 2 3 4 5"

```

## AND

```
<AND expressions...>
```

Boolean AND. Requires that all expressions evaluate to true to return true. Exits on the first expression that evaluates to false (rest of expressions are not evaluated).

Because 0 is considered false and all other values are considered true inside a routine AND returns 0 if one expression is false or the value of the last expression if all expressions are true.

Example:

```

<AND <=? 1 1> <N=? 1 2>>          --> True
<AND <=? 1 2> <SET X 2>>          --> X never set to 2 because
                                     first predicate evaluates
                                     to false
<SET X <AND 1 2 3 0 4>>            --> X is set to 0
<SET X <AND 1 4 3 2>>            --> X is set to 2

```

## APPLY

```
<APPLY routine values...>
```

Call the routine with values. <APPLY routine values ...> is equivalent to <routine values ...>, but APPLY is often used when the routine to be called is resolved during run-time (dispatch-table).

Examples:

```

<GLOBAL MYROUTINES <LTABLE ROUTINE1 ROUTINE2>>
...
<APPLY <GET ,MYROUTINES 1> .X>      --> <ROUTINE1 .X>
<APPLY <GET ,MYROUTINES 2> .X>      --> <ROUTINE2 .X>

<APPLY <GETP .OBJECT ,P?ACTION>>    --> Call ACTION-routine
                                     on OBJECT

```

## ASH, ASHIFT

```

<ASH number places>
<ASHIFT number places>          ;"Alternative syntax"

```

<b>Zapf syntax</b>	<b>Inform syntax</b>
ASHIFT	art_shift

Versions: 5-

Arithmetic shift. Shift number left when places is positive and right if it is negative. When right shift the sign is preserved (if bit 15 is 1 a 1 is shifted in, otherwise a 0 is shifted in).

```
1000 0000 0000 1010      --> 1100 0000 0000 0101
```

Also see LSH.

Examples:

```
<ASH 4 1>      --> 8
<ASH 4 -2>     --> 1
```

## ASSIGNED?

```
<ASSIGNED? Name>
```

**Zapf syntax**

```
ASSIGNED?
```

**Inform syntax**

```
check_arg_count
```

Versions: 5-

Predicate. Can test if an optional argument named name is supplied in call to routine.

Example:

```
<ROUTINE TEST("OPT" X)
<COND (<ASSIGNED? X>
      <TELL "X is assigned." CR>
)
(ELSE
  <TELL "X is not assigned." CR>
)>
>

<TEST>      --> X is not assigned.
<TEST 1>    --> X is assigned.
```

## BACK

```
<BACK table [bytes]>
```

Return table with address moved bytes back. If the count moves past the start of the table no error is raised. Default value for bytes is 1.

Note that this is not a copy of the table, it is pointing to the same table with another starting address.

Also see GET, GETB, PUT, PUTB and REST.

Example:

```
<GLOBAL TBL1 <TABLE 1 2 3 4>>      --> TBL1 = [1 2 3 4]
<GLOBAL TBL2 <REST ,STRUCT1 4>>    --> TBL2 = [3 4]
                                     Move 4 because
                                     WORD-table!
```

<SETG TBL2 <BACK ,TBL2 2>>                      --> TBL2 = [2 3 4]

## BAND, ANDB

<BAND numbers ...>  
<ANDB numbers ...>                      ;"Alternative syntax"

<b>Zapf syntax</b>	<b>Inform syntax</b>
BAND	and

All versions

Bitwise AND.

Examples:

<BAND 33 96>	--> 32
<BAND 33 96 64>	--> 0

## BCOM

<BCOM value>

<b>Zapf syntax</b>	<b>Inform syntax</b>
BCOM	not

All versions

Bitwise NOT. Reverse all bits in the WORD value (16 bits).

Examples:

<BCOM #2 000011110001111>                      --> #2 1111000011110000

## BIND

<BIND [activation] (bindings...) expressions...>

BIND defines a program block with its own set of bindings. BIND is similar to PROG but BIND doesn't create a default activation at the start of the block. If an activation is needed it must be specified. AGAIN and RETURN without specified activation inside a BIND-block will start over or return from the previous activation (most probably the ROUTINE).

Also see AGAIN, DO, PROG, REPEAT and RETURN for more details how to control program flow.

Example:

```
<ROUTINE TEST-BIND-1 ("AUX" X)
  <TELL "START ">
  <SET X 1>
  <BIND (X)
    <SET X 2>
    <TELL N .X " ">                      ;"--> 2 (Inner X)"
  >
```

```

        <TELL N .X " ">                                ;"--> 1 (Outer X)"
        <TELL "END" CR>
    >
    --> "START 2 1 END"
    <ROUTINE TEST-BIND-2 ()
        <TELL "START ">
        <BIND (X)
            <SET X <+ .X 1>>
            <TELL N .X " ">
            <COND (<=? .X 3> <RETURN>)> ;"--> exit routine"
            <AGAIN>                                ;"--> top of routine"
        >
        <TELL "END" CR>                                ;"Never reached"
    >
    --> "START 1 START 2 START 3 "

```

## BOR, ORB

```

<BOR numbers ...>
<ORB numbers ...>                                ;"Alternative syntax"

```

<b>Zapf syntax</b>	<b>Inform syntax</b>
BOR	or

All versions

Bitwise OR.

Examples:

```

<BOR 33 96>      --> 97
<BOR 33 96 64> --> 97

```

## BTST

```

<BTST value1 value2>

```

<b>Zapf syntax</b>	<b>Inform syntax</b>
BTST	test

All versions

Predicate. Binary test. Evaluates to true if all value2 bits are set in value1. Could be expressed as <=? <BAND value1 value2> value2>.

Examples:

```

<BTST 64 64>      --> TRUE
<BTST 64 63>      --> FALSE
<BTST 97 33>      --> TRUE

```

## BUFOUT

<BUFOUT value>

### **Zapf syntax**

BUFOUT

### **Inform syntax**

buffer\_mode

Versions: 4-

Flag that controls if output is buffered (to enable proper word-wrap). Value can be true or false.

Examples:

```
<BUFOUT <>>    --> Turns off buffering (disables word-wrap)
<BUFOUT T>      --> Turns on buffering
```

## CATCH

<CATCH>

### **Zapf syntax**

CATCH

### **Inform syntax**

catch

Versions: 5-

Used in conjunction with THROW. CATCH returns the current state of the stack (the "stack frame"). Also see THROW.

Example:

```
<SETG CATCH-POINT <CATCH>>    --> Saves the current stack
                                frame in global variable
```

## CHECKU

<CHECKU character>

### **Zapf syntax**

CHECKU

### **Inform syntax**

check\_unicode

Versions: 5-

Checks if a given unicode character can be printed and/or received from the keyboard. Return is in bit 0 and 1 so the return result is either 0, 1, 2 or 3.

0 = character can not be printed and not received from keyboard

1 = character can be printed but not received from keyboard

2 = character can not be printed but received from keyboard

3 = character can both be printed and received from keyboard

Example:

```
<CHECKU 65>    --> 3
```

## CLEAR

```
<CLEAR window-number>
```

### **Zapf syntax**

```
CLEAR
```

### **Inform syntax**

```
erase_window
```

Versions: 4-

Clears window with given window-number. If window-number is -1 it unsplit all windows and then clears the resulting window. If window-number is -2 it clears all windows without unsplitting.

Example:

```
<CLEAR 0>          --> Clears window 0 (the "main"-window)
```

## COLOR

```
<COLOR fg-color bg-color>
```

```
; "Version 5"
```

```
<COLOR fg-color bg-color [window-number]>
```

```
; "Versions: 6-"
```

### **Zapf syntax**

```
COLOR
```

### **Inform syntax**

```
set_colour
```

Versions: 5-

Print text in given fg-color and bg-color from this point on (flushing out text in buffer in old colors first). Version 6 supports a third argument, window-number. The colors available (if interpreter supports it) are:

0	Current color
1	Default color
2	Black
3	Red
4	Green
5	Yellow
6	Blue
7	Magenta
8	Cyan
9	White

Example:

```
<COLOR 2 9>          --> Set black text against white background
```

## COND

```
<COND (condition expressions...)...>
```

Test condition (predicate) and if condition evaluates to true expressions are executed.

IF-THEN style:

```
<COND (<AND <=? 1 1> <=? 2 2>> <TELL "IF-THEN <...>">
```

IF-THEN-ELSE style:

```
<COND (<AND <=? 1 1> <=? 2 2>>
  <TELL "THEN <...>" CR>
)
(ELSE
  <TELL "ELSE <...>" CR>
); "Or T"
)>
```

COND evaluates each condition in turn and executes the expressions directly after the first condition that evaluates to true. ELSE is an alias for T so if the first condition is false the second is always true and is executed.

SWITCH style:

```
<COND
  (<=? .SWITCH 1>
    <TELL "Variable SWITCH = 1" CR>)
  (<=? .SWITCH 2>
    <TELL "Variable SWITCH = 2" CR>)
  (<=? .SWITCH 3>
    <TELL "Variable SWITCH = 3" CR>)
  (T
    <TELL "Variable SWITCH not in (1 2 3)" CR>)
>
```

Note that only one of the (conditions expressions ...) is executed, the conditions after a condition that evaluates to true is skipped.

```
<COND
  (T
    <TELL "Variable SWITCH not in (1 2 3)" CR>)
  (<=? .SWITCH 1>
    <TELL "Variable SWITCH = 1" CR>)
  (<=? .SWITCH 2>
    <TELL "Variable SWITCH = 2" CR>)
  (<=? .SWITCH 3>
    <TELL "Variable SWITCH = 3" CR>)
>
```

In this case conditions for 1, 2 & 3 is never executed and should result in a compiler warning.

## COPYT



<COPYT src-table dest-table length>

**Zapf syntax**

COPYT

**Inform syntax**

copy\_table

Versions: 5-

Copies length number of bytes from src-table to dest-table. The tables are allowed to overlap. If length is positive then the copy is done without corrupting the src-table. If length is negative the copy is always forward from src-table to dest-table (the absolute length number of bytes) even if this corrupts src-table.

Example:

```
<GLOBAL TABLE1 <TABLE 1 2 3>>
<GLOBAL TABLE2 <TABLE 0 0 0>>
<ROUTINE TEST-COPYT()
    <COPYT ,TABLE1 ,TABLE2 6>
    <GET ,TABLE2 2>
>

<TEST-COPYT>    -->    3
```

## CRLF

<CRLF>

**Zapf syntax**

CRLF

**Inform syntax**

new\_line

All versions

Prints carriage return and line feed.

Example:

```
<CRLF>    -->    Moves cursor to position 1 on new line
```

## CURGET

<CURGET table>

**Zapf syntax**

CURGET

**Inform syntax**

get\_cursor

Versions: 4-

CURGET puts current cursor row in record 0 and current cursor column in record 1 of the supplied table. Both row and column are WORD (16-bit).

Example:

```
<GLOBAL CURTABLE <TABLE 0 0>>
```

```
<ROUTINE TEST-CURGET ()
    <CURGET ,CURTABLE>
>
```

```
<TEST-CURGET> --> Puts current row and column in CURTABLE
```

## CURSET

```
<CURSET row column> ;"Versions: 4-5"
<CURSET row column [window-number]> ;"Versions: 6-"
```

Versions: 4-

CURSET moves cursor to row and column in current window (or supplied window-number).

In versions 4-5 it is only possible to move the cursor in the upper window (window-number = 1).

In versions 6-, if row is -1 then the cursor is turned off (-2 turns it back on).

Example:

```
<CURSET 1 1> --> Move cursor to upper left corner in
                  current window
```

## DCLEAR

```
<DCLEAR picture-number [row] [column]>
```

**Zapf syntax**  
DCLEAR

**Inform syntax**  
erase\_picture

Versions: 6-

Clears (draw background color) area covered by picture-number, starting at row and column. Also see DISPLAY.

Example:

```
<DCLEAR 1 1 1> --> Clears picture 1
```

## DEC

```
<DEC name>
```

**Zapf syntax**  
DEC

**Inform syntax**  
dec

All versions

Decrease variable (signed) name with 1.

Example:

```
<ROUTINE TEST-DEC (X) <DEC .X>>
```

```
<TEST-DEC 45>      -->  44
<TEST-DEC 0>       --> -1
```

## DIRIN

```
<DIRIN stream-number>
```

<b>Zapf syntax</b>	<b>Inform syntax</b>
DIRIN	input_stream

All versions

Select input stream. Only stream-number 0 and 1 are valid.

0	Keyboard
1	File on host

Example:

```
<DIRIN 0>          -->  True and select input stream keyboard
```

## DIROUT

```
<DIROUT stream-number [table]>          ;"Versions -5"
<DIROUT stream-number [table] [width]>  ;"Versions 6-"
```

<b>Zapf syntax</b>	<b>Inform syntax</b>
DIROUT	output_stream

Directs output to one or more output streams (multiple streams can be active simultaneously). Turn on stream with positive stream-number and turn off stream with negative stream-number.

If stream 3 is active a table must be supplied. WORD 0 in table holds number of printed characters and byte 2 onward holds the characters printed. DIROUT can overrun table if not enough space is allocated.

Later versions can format output text to width (number of characters if width is positive or number of pixels if width is negative).

1	Screen
2	File on host (transcript)
3	Table
4	File of commands on host

Example:

```
<DIROUT 3>          -->  Turns on output to file
<DIROUT -3>         -->  Turns off output to file
```

## DISPLAY

```
<DISPLAY picture-number [row] [column]>
```

**Zapf syntax**

DISPLAY

**Inform syntax**

draw\_picture

Versions: 6-

Draw picture-number at coordinates row and column. If row and column are omitted the current cursor position is used.

Example:

```
<DISPLAY 1>      --> Draws picture 1 at current cursor position
```

**DLESS?**

```
<DLESS? name value>
```

**Zapf syntax**

DLESS?

**Inform syntax**

dec\_chk

All versions

Predicate. Decrease variable (signed) name with 1 and returns true if variable name is lower than value, otherwise returns false.

Example:

```
<ROUTINE TEST-DLESS? (X)
  <PRINTN <DLESS? X 100>>
  <CRLF>
  <PRINTN .X>
>

<TEST-DLESS? 101>      -->  "0\n100"
```

**DO**

```
<DO (name start end [step])
  [(END expressions ...)] expressions ...>
```

A quirk of the DO statement, which can be thought of as a cross between a Pascal-style "for" statement and a C-style "for" statement.

Pascal-style "for" statements loop over a range of values:

```
// Pascal
for i := 1 to 10 do ...
for j := 10 downto 1 do ...

// ZIL
<DO (I 1 10) ...>
<DO (J 10 1 -1) ...>
```

C-style "for" statements initialize some state, then mutate it and repeat until a condition becomes false. In ZIL, the condition is reversed - the loop exits when it becomes true:

```
// C
for (i = first(obj); i; i = next(i)) { ... }

// ZIL
<DO (I <FIRST? .OBJ> <NOT .I> <NEXT? .I>) ...>
```

Notice that every Pascal-style loop can be transformed into a C-style loop:

```
// Pascal-style loops
<DO (I 1 10) ...>
<DO (J 10 1 -1) ...>

// C-style equivalents
<DO (I 1 <G? .I 10> <+ .I 1>) ...>
<DO (J 10 <L? .J 1> <- .J 1>) ...>
```

The quirk is that the behavior of DO depends on the syntax you use for each part.

If the third value inside the parens is a complex FORM -- meaning one that isn't a simple LVAL or GVAL, like '.MAX' is -- it's assumed to be a "C-style" exit condition, otherwise it's assumed to be a "Pascal-style" upper/lower bound. Likewise, the optional fourth value is treated as either a C-style mutator or a Pascal-style step size.

More of the DO statement's quirks are demonstrated here:

```
<ROUTINE GO ()
  <TEST-PASCAL-STYLE>
  <TEST-C-STYLE>
  <TEST-MIXED-STYLE>
  <QUIT>>

<CONSTANT C-ONE 1>
<CONSTANT C-TEN 10>

<ROUTINE TEST-PASCAL-STYLE ("AUX" (ONE 1) (TEN 10))
  <TELL "== Pascal style ==" CR>

  <TELL "Counting from 1 to 10...">
  ;"1 2 3 4 5 6 7 8 9 10"
  <DO (I 1 10)
    (END <CRLF>)
    <TELL " " N .I>>

  <TELL "Counting from 1 to 10 with step 2...">
  ;"1 3 5 7 9"
  <DO (I 1 10 2)
    (END <CRLF>)
    <TELL " " N .I>>

  <TELL "Counting from 10 to 1...">
  ;"10 9 8 7 6 5 4 3 2 1"
  <DO (I 10 1)
    (END <CRLF>)>
```

```

        <TELL " " N .I>>

<TELL "Counting from 10 to 1 with step -2...">
;"10 8 6 4 2"
<DO (I 10 1 -2)
    (END <CRLF>)
    <TELL " " N .I>>

<TELL "Counting from .ONE to .TEN...">
;"1 2 3 4 5 6 7 8 9 10"
<DO (I .ONE .TEN)
    (END <CRLF>)
    <TELL " " N .I>>

<TELL "Counting from .TEN to .ONE...">
;"10"
;"Since the loop bounds aren't FIXes (numeric
literals), ZILF doesn't know the loop is meant
    to count down, and it compiles a loop that counts
up and exits after the first iteration. A DO loop
whose condition is a constant or simple FORM always
runs at least once."
    <DO (I .TEN .ONE)
        (END <CRLF>)
        <TELL " " N .I>>

<TELL "Counting from 10 to .ONE...">
;"10"
;"See above."
<DO (I 10 .ONE)
    (END <CRLF>)
    <TELL " " N .I>>

<TELL "Counting from .TEN to 1...">
;"10"
;"See above."
<DO (I .TEN 1)
    (END <CRLF>)
    <TELL " " N .I>>

<TELL "Counting from .TEN to .ONE with step -1...">
;"10 9 8 7 6 5 4 3 2 1"
<DO (I .TEN .ONE -1)
    (END <CRLF>)
    <TELL " " N .I>>

<TELL "Counting from ,C-TEN to ,C-ONE...">
;"10"
;"Even defining the loop bounds as CONSTANTS won't

```

```

tell ZILF that the loop needs to run backwards."
  <DO (I ,C-TEN ,C-ONE)
    (END <CRLF>)
    <TELL " " N .I>>

    <TELL "Counting from %,C-TEN to %,C-ONE...">
    ;"10 9 8 7 5 4 3 2 1"
    ;"The % forces ,C-TEN to be evaluated at read time,
so the loop bounds are specified as FIXes, allowing
ZILF to determine that the loop runs backwards."
    <DO (I %,C-TEN %,C-ONE)
      (END <CRLF>)
      <TELL " " N .I>>

  <CRLF>>

<OBJECT DESK
  (DESC "desk")>

<OBJECT MONITOR
  (DESC "monitor")
  (LOC DESK)>

<OBJECT KEYBOARD
  (DESC "keyboard")
  (LOC DESK)>

<OBJECT MOUSE
  (DESC "mouse")
  (LOC DESK)>

<ROUTINE TEST-C-STYLE ()
  <TELL "== C style ==" CR>

  <TELL "Counting from 10 down to 1...">
  ;"10 9 8 7 6 5 4 3 2 1"
  <DO (I 10 <L? .I 1> <- .I 1>)
    (END <CRLF>)
    <TELL " " N .I>>

  <TELL "Counting from 10 up (!) to 1...">
  ;""
  ;"Nothing is printed, because the exit condition
is initially true. A DO loop whose condition is
a complex FORM can exit before the first iteration."
  <DO (I 10 <G? .I 1> <+ .I 1>)
    (END <CRLF>)
    <TELL " " N .I>>

```

```

    <TELL "On the desk:">
    ;"monitor mouse keyboard"
    <DO (I <FIRST? ,DESK> <NOT .I> <NEXT? .I>)
        (END <CRLF>)
        <TELL " " D .I>>

    <CRLF>>

    <ROUTINE TEST-MIXED-STYLE ()
        <TELL "== Mixed ==" CR>

        <TELL "Powers of 2 up to 1000:">
        ;"1 2 4 8 16 32 64 128 256 512"
        <DO (I 1 1000 <* .I 2>)
            (END <CRLF>)
            <TELL " " N .I>>

    <CRLF>>

```

#### Highlights:

- Loops can include subsequent code in an (END ...) clause for brevity, e.g. to print a newline after a list.

A Pascal-style DO can *sometimes* determine when it needs to run backwards, even if no step size is provided.

Pascal and C style can be mixed in the same loop, e.g. <DO (I 1 1000 <\* .I 2>) ...> to count powers of 2 up to 1000.

## ERASE

```
<ERASE value>
```

#### **Zapf syntax**

```
ERASE
```

#### **Inform syntax**

```
erase_line
```

Versions: 4-

Versions 4 and 5: if the value is 1, erase from the current cursor position to the end of its line in the current window. If the value is anything other than 1, do nothing.

Version 6: if the value is 1, erase from the current cursor position to the end of its line in the current window. If not, erase the given number of pixels minus one across from the cursor (clipped to stay inside the right margin). The cursor does not move.

Example:

```
<ERASE 1>      -->  Clears from cursor to end of line
```

## F?

```
<F? expression>
```



Predicate. Test if expression evaluates to false.

Example:

```
<F? <=? 1 1>>      -->  False
<F? <=? 1 2>>      -->  True
```

## FCLEAR

```
<FCLEAR object flag>
```

### **Zapf syntax**

FCLEAR

### **Inform syntax**

clear\_attr

All versions

Removes flag from object.

Example:

```
<FCLEAR ,TRAP-DOOR ,OPENBIT>  -->  Marks the trap-door as
                                     closed
```

## FIRST?

```
<FIRST? object>
```

### **Zapf syntax**

FIRST?

### **Inform syntax**

get\_child

All versions

Returns the first object inside (contained) in the object. Returns 0 (false) if no object exists.

Example:

```
<SET RM <FIRST? ,ROOMS>>  -->  Sets RM to first object in
                                   ROOMS. Also evaluates to
                                   true (all values not 0 is true)
```

## FONT

```
<FONT number>                                ;"Version 5"
<FONT number [window-number]>                ;"Versions 6-"
```

### **Zapf syntax**

FONT

### **Inform syntax**

set\_font

Versions: 5-

Sets current font to number. Returns old fonts number. If the font number is not available 0 (false) is returned.

1	Normal font
3	Character graphics font (see §16 in <i>The Z-Machine Standards Document</i> )
4	Monospace (fixed-pitch) font

Example:

```
<FONT 4>  -->  Sets fixed-pitch font. In version 3-4 this is
                done by setting bit 1 of Flags 2 in header
                <PUT 0 8 <BOR <GET 0 8> 2>>
```

## FSET

<FSET object flag>

### **Zapf syntax**

FSET

### **Inform syntax**

set\_attr

All versions

Add flag to object.

Example:

```
<FSET ,TRAP-DOOR ,OPENBIT>  -->  Marks the trap-door as
                                   open
```

## FSET?

<FSET? object flag>

### **Zapf syntax**

FSET?

### **Inform syntax**

test\_attr

All versions

Predicate. Tests if the flag is set on the object.

Example:

```
<FSET? ,TRAP-DOOR ,OPENBIT>  -->  True if OPENBIT is set
```

## FSTACK

<FSTACK number [stack]>

### **Zapf syntax**

FSTACK

### **Inform syntax**

pop / pop\_stack

Versions: 6-

Removes number of items from system stack or given stack (table).

Example:

```
<PUSH 123> <PUSH 0> <PUSH 0> <PUSH 0> <FSTACK 3> <POP>
---> 123
```

## G?, GRTR?

```
<G? value1 value2>
<GRTR? Value1 value2>           ;Alternative syntax"
```

### **Zapf syntax**

GRTR?

### **Inform syntax**

Jg

All versions

Predicate. Returns true if value1 is greater than value2, otherwise false.

Examples:

```
<G? 5 4>  -->  T
<G? 4 5>  -->  <>
```

## G=?

```
<G=? value1 value2>
```

Predicate. Returns true if value1 is greater or equal to value2, otherwise false.

Examples:

```
<G=? 5 4>  -->  T
<G=? 5 5>  -->  T
```

## GET

```
<GET table offset>
```

### **Zapf syntax**

GET

### **Inform syntax**

loadw

All versions

Returns WORD-record (2 bytes) stored at offset.

Note: table is an address in memory so the WORD that is returned is at table+offset\*2. It is legal to use, for example, 0 as an address to retrieve information from the header.

Also see BACK, GETB, PUT, PUTB and REST.

Example:

```
<GET <TABLE 0 1 2 3> 2>           -->  2
```

## GETB

```
<GETB table offset>
```

**Zapf syntax**

GETB

**Inform syntax**

loadb

All versions

Returns BYTE-record (1 byte) stored at offset.

Note: table is an address in memory so the BYTE that is returned is at table+offset. It is legal to use, for example, 0 as an address to retrieve information from the header.

Also see BACK, GET, PUT, PUTB and REST.

Example:

```
<GETB <TABLE (BYTE) !\A !\B !\C !\D> 2>      -->  !\C
```

**GETP**

```
<GETP object property>
```

**Zapf syntax**

GETP

**Inform syntax**

get\_prop

All versions

Get property from the object. Returns default value if property is not declared in the object.

Example:

```
<OBJECT MYOBJ (MYPROP 123)>

<GETP ,MYOBJ ,P?MYPROP>  -->  123
```

**GETPT**

```
<GETPT object property>
```

**Zapf syntax**

GETPT

**Inform syntax**

get\_prop\_addr

All versions

Get property address from object. Returns 0 (false) if property is not declared in the object.

Example:

```
<OBJECT MYOBJ (MYPROP 123)>

<GET <GETPT ,MYOBJ ,P?MYPROP> 0>  -->  123
<GETPT ,MYOBJ ,P?MYPROP2>        -->  0
```

## GVAL

```
<GVAL name>  
,name ;Alternative syntax"
```

Get value of global variable name. More often used in its short form ", name".

Example:

```
<GLOBAL X 5>  
  
<GVAL X>  -->  5  
,X        -->  5
```

## HLIGHT

```
<HLIGHT style>
```

### **Zapf syntax**

HLIGHT

### **Inform syntax**

set\_text\_style

Versions: 4-

Set text to **style**. It is possible to combine styles.

0	Normal
1	Inverse
2	Bold
4	Italic
8	Monospace

Example:

```
<HLIGHT 2>          -->  Set font to bold
```

## IFFLAG

```
<IFFLAG (compilation-flag-condition expressions...)...>
```

## IGRTR?

```
<IGRTR? name value>
```

### **Zapf syntax**

IGRTR?

### **Inform syntax**

inc\_chk

All versions

Predicate. Increase variable (signed) name with 1 and returns true if variable name is lower than value, otherwise returns false.

Example:

```
<ROUTINE TEST-IGRTR? (X)
  <PRINTN <IGRTR? X 100>>
  <CRLF>
  <PRINTN .X>
>

<TEST-IGRTR? 100>  -->  "1\n101"
<TEST-IGRTR? 99>  -->  "0\n100"
```

## IN?

```
<IN? object1 object2>
```

<b>Zapf syntax</b>	<b>Inform syntax</b>
IN?	jin

All versions

Predicate. Returns true if object1 is in object2 (object1 has object2 as parent), otherwise false.

Example:

```
<OBJECT ANIMAL>
<OBJECT CAT (LOC ANIMAL)>

<IN? ,CAT ,ANIMAL>  -->  T
<IN? ,ANIMAL ,CAT>  -->  <>
```

## INC

```
<INC name>
```

<b>Zapf syntax</b>	<b>Inform syntax</b>
INC	inc

All versions

Increment name by 1. (This is signed, so -1 increments to 0)

Example:

```
<GLOBAL X 5>

<INC ,X>  -->  X=6
```

## INPUT

```
<INPUT 1 [time] [routine]>
```

<b>Zapf syntax</b>	<b>Inform syntax</b>
--------------------	----------------------

INPUT                      read\_char

Versions: 4-

INPUT reads a single character from the keyboard. Calls routine every time\*0.1 s. If routine returns true input is aborted.

Examples:

```
<INPUT 1> --> Wait for keypress
```

```
<ROUTINE WAIT-TWO-SECONDS ()  
  <INPUT 1 20 ABORT-WAIT>  
>
```

```
<ROUTINE ABORT-WAIT () <RETURN T>>
```

```
<WAIT-TWO-SECONDS> --> Pause two seconds (if not  
                        interrupted by a keypress  
                        from the keyboard
```

## INTBL?

```
<INTBL? value table length [form]>            ;"Version 5"  
<INTBL? value table length>                 ;"Version 4, 6-"
```

<b>Zapf syntax</b>	<b>Inform syntax</b>
INTBL?	scan_table

Versions: 4-

Predicate. Returns value if value is in table of length, otherwise 0.

In version 5 the form describes the field where bit 7 is set for words and clear for bytes, rest defines the length of the field.

Examples:

```
<INTBL? 3 <TABLE 1 2 3 4> 4> --> 3  
<INTBL? 6 <TABLE 1 2 3 4> 4> --> 0  
<INTBL? 8 <TABLE (BYTE) 2 0 1 4 0 1 8 0 1> 9 3>    --> 8  
                                                     ;"Ver 8"
```

## IRESTORE

```
<IRESTORE>
```

<b>Zapf syntax</b>	<b>Inform syntax</b>
IRESTORE	restore_undo

Versions: 5-

Restores game state saved to memory by ISAVE (undo).

## ISAVE

<ISAVE>

### Zapf syntax

ISAVE

### Inform syntax

save\_undo

Versions: 5-

Save game state to memory that later can be restored by IRESTORE (undo). Returns 0 if ISAVE fails, 1 if it is successful and -1 if the interpreter does not handle undo.

## ITABLE

<ITABLE [specifier] count [(flags...)] defaults ...>

Defines a table of `count` elements filled with default values: either zeros or, if the `default` list is specified, the specified list of values repeated until the table is full.

The optional `specifier` may be the atoms `NONE`, `BYTE`, or `WORD`. `BYTE` and `WORD` change the type of the table and also turn on the length marker (element 0 in the table contains the length of the table), This can also be done with the flags (see `TABLE` about flags).

Examples:

<ITABLE 4 0> -->

Element 0 WORD	Element 1 WORD	Element 2 WORD	Element 3 WORD
0	0	0	0

<ITABLE (BYTE LENGTH) 4 0> -->

Element 0 BYTE	Element 1 BYTE	Element 2 BYTE	Element 3 BYTE	Element 4 BYTE
4	0	0	0	0

<ITABLE BYTE 4 0> -->

Element 0 BYTE	Element 1 BYTE	Element 2 BYTE	Element 3 BYTE	Element 4 BYTE
4	0	0	0	0

## L?, LESS?

<L? value1 value2>

<LESS? Value1 value2> ;Alternative syntax"

### Zapf syntax

LESS?

### Inform syntax

J1



All versions

Predicate. Returns true if value1 is less than value2, otherwise false.

Examples:

```
<L? 5 4> --> <>
<L? 4 5> --> T
```

## L=?

```
<L=? value1 value2>
```

Predicate. Returns true if value1 is less or equal to value2, otherwise false.

Examples:

```
<L=? 5 4> --> <>
<L=? 5 5> --> T
```

## LEX

```
<LEX text parse [dictionary] [flag]>
```

**Zapf syntax**

LEX

**Inform syntax**

tokenise

Versions: 4-

Parse the text into parse. See READ for more info about parsing. The game dictionary is used if not a dictionary table (LTABLE) is supplied. If the length of the dictionary is negative, the dictionary can be unsorted. If the flag is set (true), unrecognized words are not written to parse but their slot is left unmodified. This makes it possible to run LEX against different dictionaries serially. Also see READ.

Example:

```
<GLOBAL TEXTBUF <TABLE (BYTE) !\c !\a !\t>>
<GLOBAL PARSEBUF <ITABLE 1 (LEXV) 0 0>>
<OBJECT CAT (SYNONYM CAT)>

<LEX ,TEXTBUF ,PARSEBUF>
<PRINTB <GET ,PARSEBUF 1>> --> "cat"
```

## LOC

```
<LOC object>
```

**Zapf syntax**

LOC

**Inform syntax**

get\_parent

All versions

Returns parent to object.

Examples:

```
<OBJECT ANIMAL>
<OBJECT CAT (LOC ANIMAL)>

<=? <LOC ,CAT> ,ANIMAL> --> T
<LOC ,ANIMAL> --> 0
```

## LOWCORE-TABLE

```
<LOWCORE-TABLE field-spec length routine>
```

LOWCORE-TABLE reads the length number of bytes from field-spec and calls routine between each byte. See appendix B for list of valid values for field-spec.

Example:

```
<LOWCORE-TABLE SERIAL 6 PRINTC> --> Reads 6 bytes from
                                     SERIAL and print each
                                     byte as character
```

## LOWCORE

```
<LOWCORE field-spec [new-value]>
```

LOWCORE reads and in some cases writes to the header information fields. See appendix B for list of valid values for field-spec.

Examples:

```
<LOWCORE FLAGS <BOR <LOWCORE FLAGS> 2>> -->
Monospace bit (bit 1) in flags 2 is set
<PUT 0 8 <BOR <GET 0 8> 2>> --> Do the same as above
<PRINTN <BAND <LOWCORE RELEASEID> *3777*>>
--> Print the 11 lower bytes in releaseid
```

## LSH, SHIFT

```
<LSH number places>
<SHIFT number places> ;Alternative syntax"
```

<b>Zapf syntax</b>	<b>Inform syntax</b>
SHIFT	log_shift

Versions: 5-

Bitwise shift. Shift number left when places is positive and right if it is negative. When right shifting the sign is not preserved (0 is always shifted in).

```
1000 0000 0000 1010 --> 0100 0000 0000 0101
```

Also see ASH.

Examples:

```
<LSH 4 1>      --> 8
<LSH 4 -2>     --> 1
```

## LTABLE

```
<LTABLE [(flags ...)] values ...>
```

Defines a table containing the specified values and with the LENGTH flag (see TABLE about LENGTH and other flags).

## LVAL

```
<LVAL name>
.name                ;Alternative syntax"
```

Get value of local variable name. More often used in its short form ".name".

Example:

```
<SET X 5>
<LVAL X>  --> 5
.X        --> 5
```

## MAP-CONTENTS

```
<MAP-CONTENTS (name [next] object)
  [(END expressions ...)] expressions ...>
```

Loop over all objects that have an object as parent (all children to object). For each iteration name is assigned the current child-object and next the child-object that will be name in the next iteration (0 if current name is the last child).

For each iteration the expressions are evaluated and, if supplied, the (END expressions ...) is evaluated last after all iterations.

Example:

```
<OBJECT SURVIVAL-KIT
  (DESC "adventure survival kit") (WEIGHT 10)>
<OBJECT SWORD
  (IN SURVIVAL-KIT) (DESC "sword") (WEIGHT 10)>
<OBJECT LAMP
  (IN SURVIVAL-KIT) (DESC "brass lamp") (WEIGHT 5)>
<OBJECT SPOON
  (IN SURVIVAL-KIT) (DESC "chrome spoon") (WEIGHT 2)>

<ROUTINE TEST-MAP-CONTENTS ()
  <TELL "Your " D ,SURVIVAL-KIT " contains:" CR>
  <MAP-CONTENTS (F ,SURVIVAL-KIT)
    <TELL "      a " D .F CR>
  >
```

```

<TELL "Your " D ,SURVIVAL-KIT " contains:" CR>
<MAP-CONTENTS (F N ,SURVIVAL-KIT)
  <TELL "      a " D .F >
  <COND (.N <TELL " (next item is the " D .N ")")>>
  <TELL CR>
>

<BIND ((W 0))
  <SET W <GETP ,SURVIVAL-KIT ,P?WEIGHT>>
  <MAP-CONTENTS (F ,SURVIVAL-KIT)
    (END <TELL "Total weight is = " N .W CR>)
    <SET W <+ .W <GETP .F ,P?WEIGHT>>>
  >
>

<TEST-MAP-CONTENTS>      -->
Your adventure survival kit contains:
  a sword
  a chrome spoon
  a brass lamp
Your adventure survival kit contains:
  a sword (next item is the chrome spoon)
  a chrome spoon (next item is the brass lamp)
  a brass lamp
Total weight is = 27

```

## MAP-DIRECTIONS

```

<MAP-DIRECTIONS (name pt room)
  [(END expressions ...)] expressions ...>

```

Loop over all directions in a room. For each iteration name is assigned the current direction and pt is the room the direction leads to.

For each iteration the expressions are evaluated and, if supplied, the (END expressions ...) is evaluated last after all iterations.

Example:

```

<DIRECTIONS NORTH SOUTH EAST WEST>
<OBJECT CENTER (DESC "center room")
  (NORTH TO N-ROOM)
  (WEST TO W-ROOM)>
<OBJECT N-ROOM (DESC "north room")>
<OBJECT W-ROOM (DESC "west room")>

<ROUTINE TEST-MAP-DIRECTIONS ()
  <TELL "You're in the " D ,CENTER>
  <TELL CR "Obvious exits:" CR>
  <MAP-DIRECTIONS (D P ,CENTER)

```

```

        (END <TELL "Room description done." CR>)
    <COND (<EQUAL? .D ,P?NORTH> <TELL "    North">)
        (<EQUAL? .D ,P?SOUTH> <TELL "    South">)
        (<EQUAL? .D ,P?EAST> <TELL "    East">)
        (<EQUAL? .D ,P?WEST> <TELL "    West">)
    >
    <VERSION?
        (ZIP <TELL " to the " D <GETB .P ,REXIT> CR>)
        (ELSE <TELL " to the " D <GET .P ,REXIT> CR>)
    >
>
>

```

## MARGIN

```
<MARGIN left right [window-number]>
```

### **Zapf syntax**

MARGIN

### **Inform syntax**

set\_margins

Versions: 6-

Set left and right margin (in pixels) in the given window-number. If no window-number is specified MARGIN sets margins in window-number 0.

Example:

```
<MARGIN 1 1> --> set 1 pixel margin in window 0
```

## MENU

```
<MENU number table>
```

### **Zapf syntax**

MENU

### **Inform syntax**

make\_menu

Versions: 6-

Controls menu 3- (not menu 0-2, they are system menus). The table is a LTABLE of LTABLE. Item 1 being the menu name. Item 2- are the entries.

Example (from Journey):

```

<GLOBAL MAC-SPECIAL-MENU
    <LTABLE <TABLE (STRING LENGTH) "Journey">
        <TABLE (STRING LENGTH) "Essences">
        <TABLE (STRING LENGTH) "No Defaults">>>
    ...
<MENU 3 ,MAC-SPECIAL-MENU>

```

## MOD

<MOD number1 number2>

### **Zapf syntax**

MOD

### **Inform syntax**

mod

All versions

Returns remainder of 16-bit signed division. number2 is not allowed to be 0 ("Division by zero").

Examples:

```
<MOD 15 4>      --> 3
<MOD -15 4>     --> -3
<MOD -15 -4>    --> -3
<MOD 15 -4>     --> 3
```

## MOUSE-INFO

<MOUSE-INFO table>

### **Zapf syntax**

MOUSE-INFO

### **Inform syntax**

read\_mouse

Versions: 6-

Reads mouse information into table. The table is 4 WORDS (2 bytes) long.

0	Y coordinate
1	X coordinate
2	Button bits (host dependent)
3	Menu (number*256+entry)

Example (from Journey):

```
<GLOBAL MOUSE-INFO-TBL <TABLE 0 0 0 0>>
...
<MOUSE-INFO ,MOUSE-INFO-TBL>
```

## MOUSE-LIMIT

<MOUSE-LIMIT window-number>

### **Zapf syntax**

MOUSE-LIMIT

### **Inform syntax**

mouse\_window

Versions: 6-

Restricts mouse movement to window-number. If window-number is -1 all restrictions are removed. 1 is default window-number.

Example:

```
<MOUSE-LIMIT 1>          -->  Mouse constrained to window 1
```

## MOVE

```
<MOVE object1 object2>
```

### **Zapf syntax**

MOVE

### **Inform syntax**

insert\_obj

All versions

Move object1 to be the first child of object2. Children of object1 move with it.

Example:

```
<OBJECT ANIMAL>
```

```
<OBJECT CAT>
```

```
<MOVE ,CAT ,ANIMAL>
```

```
<IN? ,CAT ,ANIMAL>  -->  T
```

## N=?, N==?

```
<N=? value1 value2...valueN>
```

```
<N==? value1 value2...valueN> ;Alternative syntax"
```

Predicate. True if value1 is not equal to any of the values value2 to valueN.

Examples:

```
<N=? 1 1>          -->  FALSE
```

```
<N=? 1 2>          -->  TRUE
```

```
<N=? 1 2 1>        -->  FALSE
```

## NEXT?

```
<NEXT? object>
```

### **Zapf syntax**

NEXT?

### **Inform syntax**

get\_sibling

All versions

Returns object after object in object-list (sibling). Returns 0 (false) if no object exists.

Example:

```
<OBJECT ANIMAL>
```

```
<OBJECT CAT>
```

```

<OBJECT DOG>

<MOVE ,CAT ,ANIMAL>
<MOVE ,DOG ,ANIMAL>
<=? <NEXT? ,DOG> ,CAT>          -->  T

```

## NEXTP

```
<NEXTP object property>
```

<b>Zapf syntax</b>	<b>Inform syntax</b>
NEXTP	get_next_prop

All versions

Returns the property that comes after property on the object. Returns 0 if there are no more properties after property. If property is 0 then NEXTP returns first property on object.

Example:

```

<OBJECT MYOBJ (FOO 123) (BAR 456)>

<=? <NEXTP ,MYOBJ 0> P?FOO>          -->  T
<=? <NEXTP ,MYOBJ P?FOO> P?BAR>      -->  T
<NEXTP ,MYOBJ P?BAR>                  -->  0 (false)

```

## NOT

```
<NOT expression>
```

Returns the boolean NOT of expression.

Examples:

```
<NOT <=? 1 2>> -->  True (1)
```

## OR

```
<OR expressions...>
```

Boolean OR. Requires that one of the expressions evaluates to true to return true. Exits on the first expression that evaluates to true (rest of expressions are not evaluated).

Because 0 is considered false and all other values are considered true inside a routine OR returns 0 if all expressions are false or the value of the first true expression.

Example:

```

<OR <=? 1 2> <=? 1 1>>          -->  True
<OR <=? 1 1> <SET X 2>>        -->  X never set to 2 because
                                   first predicate evaluates
                                   to true
<SET X <OR 0 1 2 3>>            -->  X is set to 1
<SET X <OR 0 <> 0>>              -->  X is set to 0

```



## ORIGINAL?

<ORIGINAL?>

**Zapf syntax**  
ORIGINAL?

**Inform syntax**  
piracy

Versions: 5-

Predicate. Tests if the game disc is an original. Almost all modern interpreters always return true.

## PICINF

<PICINF picture-number table>

**Zapf syntax**  
PICINF

**Inform syntax**  
picture\_data

Versions: 6-

Writes picture data from picture-number into table. Word 0 of table holds picture width and word 1 holds picture height. Then follows the picture data.

If picture-number is 0, the number of available pictures is written into word 0 of table and release number of picture file is written into word 1.

Example:

<GLOBAL MYPIC <ITABLE 2048 0>>

<PICINFO 1 ,MYPIC> --> Writes picture data into MYPIC

## PICSET

<PICSET table>

**Zapf syntax**  
PICSET

**Inform syntax**  
picture\_table

Versions: 6-

Give the interpreter a table of picture numbers that the interpreter can then unpack from disc and cache in memory.

## PLTABLE

<PLTABLE [(flags ...)] values ...>

Defines a table containing the specified values and with the PURE and LENGTH flag (see TABLE about LENGTH, PURE and other flags).

## POP

<POP [stack]>

### **Zapf syntax**

POP

### **Inform syntax**

pull

Versions: 6-

Pops value of stack. If no stack is given, a value is popped from the game stack.

Example:

<PUSH 123>

<POP> --> 123

<GLOBAL MY-STACK <TABLE 3 0 0 123>>

<POP ,MY-STACK> --> 123

## PRINT

<PRINT packed-string>

### **Zapf syntax**

PRINT

### **Inform syntax**

print\_paddr

All versions

Print packed-string from high memory (packed address).

Example:

<GLOBAL MSG "Hello, sailor!">

<PRINT ,MSG> --> "Hello, sailor!"

## PRINTB

<PRINTB unpacked-string>

### **Zapf syntax**

PRINTB

### **Inform syntax**

print\_addr

All versions

Print unpacked-string from dynamic or static memory (unpacked address).

Example:

<OBJECT MYOBJECT (SYNONYM HELLO)>

<PRINTB <GETP ,MYOBJECT ,P?SYNONYM>> --> "hello"

## PRINTF

<PRINTF character>

### **Zapf syntax**

PRINTF

### **Inform syntax**

print\_char

All versions

Print character.

Example:

<PRINTF 65>      -->    A

## PRINTD

<PRINTD object>

### **Zapf syntax**

PRINTD

### **Inform syntax**

print\_obj

All versions

Print description of object.

Example:

<GLOBAL MYOBJECT (DESC "sword">

<PRINTD ,MYOBJECT>    -->    "sword"

## PRINTF

<PRINTF table>

### **Zapf syntax**

PRINTF

### **Inform syntax**

print\_form

Versions: 6-

Print a formatted table. Each line starts with a WORD that is the number of characters that follows. Last byte in each line is 0.

## PRINTI

<PRINTI string>

### **Zapf syntax**

PRINTI

### **Inform syntax**

print

All versions

Print string.

Example:

```
<PRINTI "Hello, sailor!">      -->  "Hello, sailor!"
```

## PRINTN

```
<PRINTN number>
```

### **Zapf syntax**

PRINTN

### **Inform syntax**

print\_num

All versions

Print number.

Example:

```
<PRINTN <+ 1 3>>      -->   4
<PRINTN -42>           --> -42
```

## PRINTR

```
<PRINTR string>
```

### **Zapf syntax**

PRINTR

### **Inform syntax**

print\_ret

All versions

Print string and then CRLF.

Example:

```
<PRINTR "Hello. Sailor!">      -->  "Hello, sailor!\n"
```

## PRINTT

```
<PRINTT table width [height] [skip]>
```

### **Zapf syntax**

PRINTT

### **Inform syntax**

print\_table

Versions: 5-

Print table (string) in rectangle defined by width and height. Default height is 1. If skip is given then that number of characters is skipped between lines.

Examples:

```
<GLOBAL MYTEXT <TABLE (STRING) "hansprestige">>
```

```
<PRINTT ,MYTEXT 6>      -->  "hanspr\n"
<PRINTT ,MYTEXT 4 3>    -->  "hans\npres\ntige\n"
<PRINTT ,MYTEXT 3 3 1>  -->  "han\npre\ntig\n"
```

## PRINTU

<PRINTU number>

### **Zapf syntax**

PRINTU

### **Inform syntax**

print\_unicode

Versions: 5-

Print unicode-character number.

Examples:

```
<PRINTU 65>          -->  A
<PRINTU 196>         -->  Ä
```

## PROG

<PROG [activation] (bindings...) expressions...>

PROG defines a program block with its own set of bindings. PROG is similar to BIND but PROG automatically creates a default activation at the start of the block which you optionally can name. This means that AGAIN moves program execution to this activation. RETURN exits this PROG-block.

Note that there is a special variable, DO-FUNNY-RETURN?, that controls how RETURN with value should be handled. If DO-FUNNY-RETURN? is true then RETURN value returns from ROUTINE, otherwise it returns from PROG. DO-FUNNY-RETURN? is default false in version 3-4 and default true in versions 5-.

Also see AGAIN, BIND, DO, REPEAT and RETURN for more details how to control program flow. AGAIN and RETURN have examples on how activation and DO-FUNNY-RETURN? works.

Examples:

```
; "Block have own set of atoms"
<ROUTINE TEST-PROG-1 ("AUX" X)
  <SET X 2>
  <TELL "START: ">
  <PROG (X)
    <SET X 1>
    <TELL N .X " ">      ; "Inner X"
  >
  <TELL N .X>            ; "Outer X"
  <TELL " END" CR CR>
>
-->  "START: 1 2 END"
```

```
; "AGAIN, Bare RETURN without ACTIVATION"
<ROUTINE TEST-PROG-2 ()
  <TELL "START: ">
  <PROG (X)  ; "X is not reinitialized between iterations."
```

```

    Default ACTIVATION created."
    <SET X <+ .X 1>>
    <TELL N .X " ">
    <COND (<=? .X 3> <RETURN>)>          ;"Bare RETURN without
                                         ACTIVATION will exit
                                         BLOCK"
    <AGAIN> ;"AGAIN without ACTIVATION will redo BLOCK"
  >
  <TELL "RETURN EXIT BLOCK" CR CR>
>
--> "START: 1 2 3 RETURN EXIT BLOCK"

;"AGAIN, RETURN with value but without ACTIVATION"
<ROUTINE TEST-PROG-3 ()
  <TELL "START: ">
  <PROG ((X 0)) ;"X is not reinitialized between
               iterations. Default ACTIVATION created."
  <SET X <+ .X 1>>
  <TELL N .X " ">
  <COND (<=? .X 3>
    <COND (,FUNNY-RETURN?
      <TELL "RETURN EXIT ROUTINE" CR CR>)>
      <RETURN T>)> ;"RETURN with value but without
                  ACTIVATION will exit ROUTINE
                  (FUNNY-RETURN = TRUE)"
  <AGAIN> ;"AGAIN without ACTIVATION will redo BLOCK"
>
  <TELL "RETURN EXIT BLOCK" CR CR>
>
--> "START: 1 2 3 RETURN EXIT ROUTINE"

```

## PTABLE

```
<PTABLE [(flags ...)] values ...>
```

Defines a table containing the specified values and with the PURE flag (see TABLE about PURE and other flags).

## PTSIZE

```
<PTSIZE property-address>
```

### **Zapf syntax**

```
PTSIZE
```

### **Inform syntax**

```
get_prop_len
```

All versions

Get size in bytes of property at property-address.

Example:

<OBJECT MYOBJECT (FOO 1 2 3)>

<PTSIZE <GETPT ,MYOBJECT ,P?FOO>> --> 6

## PUSH

<PUSH value>

### **Zapf syntax**

PUSH

### **Inform syntax**

push

All versions

Push value on game stack.

Example:

<PUSH 123>

## PUT

<PUT table offset value>

### **Zapf syntax**

PUT

### **Inform syntax**

storew

All versions

Put a 16-bit WORD value in the table at word position offset. Actual address is table-address+offset\*2.

Note that table can be a byte-address in dynamic memory.

Also see BACK, GET, GETB, PUTB and REST.

Examples:

<PUT ,MYTABLE 1 123> --> Stores 123 at position 1  
in MYTABLE

<PUT 0 8 <BOR <GET 0 8> 2>> --> Sets bit 1 in Flags 2 in  
header (force monospace)

## PUTB

<PUTB table offset value>

### **Zapf syntax**

PUTB

### **Inform syntax**

storeb

All versions

Put a byte value in the table at byte position offset. Actual address is table-address+offset.

Note that table can be a byte-address in dynamic memory.

Also see BACK, GET, GETB, PUT and REST.

Example:

```
<PUTB ,MYTABLE 1 !\A>      -->  Stores character A at
                                position 1 in MYTABLE
```

## PUTP

```
<PUTP object property value>
```

<b>Zapf syntax</b>	<b>Inform syntax</b>
PUTP	put_prop

All versions

Put value into property on the object.

Example:

```
<OBJECT MYOBJ (MYPROP 123)>

<PUTP ,MYOBJ ,P?MYPROP 456>  -->  Stores 456 in property
                                MYPROP on MYOBJ
```

## QUIT

```
<QUIT>
```

<b>Zapf syntax</b>	<b>Inform syntax</b>
QUIT	quit

All versions

Halts game execution. No questions asked.

## RANDOM

```
<RANDOM range>
```

<b>Zapf syntax</b>	<b>Inform syntax</b>
RANDOM	random

All versions

Returns a random number between 1 and range. If range is negative the randomizer is reseeded with -range (absolute value of range).

Example:

```
<- <RANDOM 101> 1>  -->  Generates random number
                        between 0-100
```



## READ

```
<READ text parse> ;"Versions 1-3"
<READ text parse [time] [routine]> ;"Version 4"
<READ text [parse] [time] [routine]> ;"Versions 5-"
```

### **Zapf syntax**

READ

### **Inform syntax**

aread / sread

All versions

Reads text from keyboard and parse it. Result is stored in two byte-tables. Byte 0 in `text` must contain the max-size of the buffer and if `parse` is supplied, byte 0 of it must contain a max number of words that will be parsed.

After READ, `text` contains:

Byte	0	Max number of chars read into the buffer
	1	Actual number of chars read into the buffer
	2-	The typed chars all converted to lowercase

`parse` contains:

Byte	0	Max number of words parsed
	1	Actual number of words parsed
	2-3	Address to first word in dictionary (0 if word is not in it)
	4	Length of first word
	5	Start position (in <code>text</code> ) of first word
	6-9	Second word
	...	

Example:

```
<GLOBAL READBUF <ITABLE BYTE 63>>
<GLOBAL PARSEBUF <ITABLE BYTE 28>>
<ROUTINE READ-TEST ("AUX" WORDS WLEN WSTART WEND)
<PUTB ,READBUF 0 60>
<PUTB ,PARSEBUF 0 6>
<READ ,READBUF ,PARSEBUF>
<SET WORDS <GETB ,PARSEBUF 1>> ;"# of parsed words"
<DO (I 1 .WORDS)
  <SET WLEN <GETB .PARSEBUF <+ .I 4>>>
  <SET WSTART <GETB .PARSEBUF <+<+ .I 4> 1>>>
  <SET WEND <+ .WSTART <- .WLEN 1>>>
  <TELL "word " N .I " is " N .WLEN " char long. ">
  <TELL "The word is '">
  <DO (J .WSTART .WEND)
    <PRINTC <GETB .READBUF .J>> ;"To lcase!"
  >
  <TELL "'.'" CR>
>
>
```

See *The Inform Designer's Manual* (ch. §2.5, p. 44-46) for more details about READ.

## REMOVE

<REMOVE object>

### **Zapf syntax**

REMOVE

### **Inform syntax**

remove\_obj

All versions

Remove object from parent. See MOVE how to reattach it to another object.

Example:

<OBJECT ANIMAL>

<OBJECT CAT (LOC ANIMAL)>

<REMOVE ,CAT>           --> Detach CAT from ANIMAL

## REPEAT

<REPEAT [activation] (bindings...) expressions...>

REPEAT defines a program block with its own set of bindings. REPEAT is very similar to PROG the only difference is that at the end of the block is an automatic AGAIN. REPEAT automatically creates a default activation at the start of the block which you optionally can name. This means that AGAIN moves program execution to this activation. RETURN exits this REPEAT-block.

Note that there is a special variable, DO-FUNNY-RETURN?, that controls how RETURN with value should be handled. If DO-FUNNY-RETURN? is true then RETURN value returns from ROUTINE, otherwise it returns from REPEAT. DO-FUNNY-RETURN? is default false in version 3-4 and default true in versions 5-.

Also see AGAIN, BIND, DO, PROG and RETURN for more details how to control program flow. AGAIN and RETURN have examples on how activation and DO-FUNNY-RETURN? works.

Examples:

```
; "Bare RETURN without ACTIVATION"
<ROUTINE TEST-REPEAT-1 ()
  <TELL "START: ">
  <REPEAT (X)       ; "X is not reinitialized between iterations.
                    Default ACTIVATION created."
    <SET X <+ .X 1>>
    <TELL N .X " ">
    <COND (<=? .X 3> <RETURN>)>       ; "Bare RETURN without
                                        ACTIVATION will exit
                                        BLOCK"
  >
  <TELL "RETURN EXIT BLOCK" CR CR>
>
```

```

-->  "START: 1 2 3 RETURN EXIT BLOCK"

;"RETURN with value but without ACTIVATION"
<ROUTINE TEST-REPEAT-2 ()
  <TELL "START: ">
  <REPEAT ((X 0)) ;"X is not reinitialized between
                    iterations. Default ACTIVATION created."
    <SET X <+ .X 1>>
    <TELL N .X " ">
    <COND (<=? .X 3>
      <COND (,FUNNY-RETURN?
        <TELL "RETURN EXIT ROUTINE" CR CR>)>
        <RETURN T>)> ;"RETURN with value but without
                    ACTIVATION will exit ROUTINE
                    (FUNNY-RETURN = TRUE)"
    >
  <TELL "RETURN EXIT BLOCK" CR CR>
>
-->  "START: 1 2 3 RETURN EXIT ROUTINE"

```

## REST

```
<REST table [bytes]>
```

Return table without its first bytes (bytes is default 1). Note that this is not a copy of the table, it is pointing to the same table with another starting address.

Also see BACK, GET, GETB, PUT and PUTB.

Example:

```

<GLOBAL TBL1 <TABLE 1 2 3 4>>      -->  TBL1 = [1 2 3 4]
<GLOBAL TBL2 <REST ,TBL1 2>>      -->  TBL2 = [2 3 4]
                                   Move 2 because
                                   WORD-table!
<PUT ,TBL2 0 5>                    -->  TBL1 = [1 5 3 4],
                                   TBL2 = [5 3 4]

```

## RESTART

```
<RESTART>
```

**Zapf syntax**

```
RESTART
```

**Inform syntax**

```
restart
```

All versions

Restarts the game. No questions asked. The only things that survive a restart are bit 0 and bit 1 of Flags 2 in the header (setting for transcribing and monospace).

## RESTORE

```
<RESTORE> ; "Versions 1-4"
<RESTORE [table] [bytes] [filename]> ; "Versions 5-"
```

<b>Zapf syntax</b>	<b>Inform syntax</b>
RESTORE	restore

All versions

RESTORE a game to a previously saved state. All questions about filename and path are asked by the interpreter.

If RESTORE fails, game execution continues with the next statement after RESTORE.

If RESTORE is successful game execution continues from where the SAVE was issued (SAVE returns 2 in this case).

See *The Inform Designer's Manual* (ch. §42, p. 319) and *The Z-machine Standards Document* for a description about how to SAVE and RESTORE auxiliary files.

Example:

```
<ROUTINE SAVE-GAME ("AUX" RESULT)
  <SET RESULT <SAVE>>
  <COND (<=? .RESULT 0> <TELL "Save failed." CR>)>
  <COND (<=? .RESULT 1> <TELL "Save successful." CR>)>
  <COND (<=? .RESULT 2> <TELL "Restore successful." CR>)>
>

<ROUTINE RESTORE-GAME ()
  <RESTORE>
  <TELL "Restore failed." CR>
>
```

## RETURN

```
<RETURN [value] [activation]>
```

<b>Zapf syntax</b>	<b>Inform syntax</b>
RETURN	ret

All versions

RETURN from current routine with value. Returns 1 (true) if no value is given.

RETURN is also used in commands that control program flow to exit program blocks. Also see AGAIN, BIND, DO, PROG and REPEAT for more details how to control program flow.

Examples:

```
<RETURN> --> Returns 1
<RETURN 42> --> Returns 42
```

## RFALSE

<RFALSE>

### **Zapf syntax**

RFALSE

### **Inform syntax**

rfalse

All versions

RFALSE always exits routine and returns false (0). Note that this differs from RETURN that can both exit program blocks and routines.

## RFATAL

<RFATAL>

RFATAL always exits routine and returns FATAL-VALUE (2). Note that this differs from RETURN that can both exit program blocks and routines.

## RSTACK

<RSTACK>

### **Zapf syntax**

RSTACK

### **Inform syntax**

ret\_popped

All versions

Pops value from game stack and returns that value.

Example:

<PUSH 42>

<RSTACK>           --> Returns 42

## RTRUE

<RTRUE>

### **Zapf syntax**

RTRUE

### **Inform syntax**

rtrue

All versions

RTRUE always exits routine and returns true (1). Note that this differs from RETURN that can both exit program blocks and routines.

## SAVE

<SAVE>

; "Versions 1-4"

<SAVE [table] [bytes] [filename]>

; "Versions 5-"

**Zapf syntax**

SAVE

**Inform syntax**

save

All versions

SAVE a game state that later can be restored. All questions about filename and path are asked by the interpreter.

SAVE returns 0 if SAVE fails and 1 if it is successful.

SAVE also can return 2. That means this is a continuation from a successful RESTORE.

See RESTORE on code example on SAVE and RESTORE.

See *The Inform Designer's Manual* (ch. §42, p. 319) and *The Z-machine Standards Document* for a description about how to SAVE and RESTORE auxiliary files.

**SCREEN**

&lt;SCREEN window-number&gt;

**Zapf syntax**

SCREEN

**Inform syntax**

set\_window

Versions: 3-

Select window-number for text output.

Note that in versions 3-5 only the lower screen (window-number = 0) has text-buffering and word-wrap.

Example:

```
<SPLIT 3>
<SCREEN 1>
<TELL "West of House">    --> Split screen in 2 (upper
                                screen is 3 rows) and write
                                "West of House" in upper screen
```

**SCROLL**

&lt;SCROLL window-number pixels&gt;

**Zapf syntax**

SCROLL

**Inform syntax**

scroll\_window

Versions: 6-

Scrolls window-number up (pixels is positive) or down (pixels is negative) the number of pixels supplied. The new lines are empty (background color).

**SET**

&lt;SET name value&gt;

**Zapf syntax**

SET

**Inform syntax**

store

All versions

Store value in local variable name.

Example:

```
<SET MYVAR 42>      -->  Store 42 in local variable MYVAR
```

**SETG**

```
<SETG name value>
```

**Zapf syntax**

SET

**Inform syntax**

store

All versions

Store value in global variable name. The name variable must be declared with GLOBAL outside the ROUTINE.

Example:

```
<SETG MYVAR 42>-->  Store 42 in global variable MYVAR
```

**SOUND**

```
<SOUND number [effect] [volrep]>          ;"Versions 3-4"
```

```
<SOUND number [effect] [volrep] [routine]> ;"Versions 5-"
```

**Zapf syntax**

SOUND

**Inform syntax**

sound\_effect

Versions: 3-

Plays sound number (1 = high-pitch beep, 2 = low-pitch beep and 3- is user defined).

Valid entries for effect are 1 = prepare, 2 = start, 3 = stop and 4 = finished with.

The volrep is calculated as 256 \* repetitions + volume. Repetitions can be 0-255 (255 = infinite) and volume 1-8, 255 (1 = quiet, 8 = loud, 255 = loudest possible).

If routine is supplied it is called after sound is finished.

See *The Inform Designer's Manual* (ch. §42, p. 315-316 and ch. §43) and *The Z-machine Standards Document* for a description about how to include sound in games.

**SPLIT**

```
<SPLIT number>
```

**Zapf syntax****Inform syntax**

SPLIT                      split\_window

Versions: 3-

SPLIT screen in two parts with the upper part having `number` rows. If `number` is 0 the screen is unsplit. The upper screen is window-number 1 and the lower screen is window-number 0.

See SCREEN for example on how to use SPLIT.

## T?

<T? expression>

Predicate. Test if `expression` evaluates to true ( not 0).

Example:

```
<T? <=? 1 1>>      -->  True
<T? <=? 1 2>>      -->  False
```

## TABLE

<TABLE [(flags ...)] values ...>

Defines a table containing the specified values.

These flags control the format of the table:

WORD causes the elements to be 2-byte words. This is the default.

BYTE causes the elements to be single bytes.

LEXV causes the elements to be 4-byte records. If `default` values are given to ITABLE with this flag, they will be split into groups of three: the first compiled as a word, the next two compiled as bytes. The table is also prefixed with a byte indicating the number of records, followed by a zero byte

STRING causes the elements to be single bytes and also changes the initializer format. This flag may not be used with ITABLE. When this flag is given, any `values` given as strings will be compiled as a series of individual ASCII characters, rather than as string addresses.

These flags alter the table without changing its basic format:

LENGTH causes a length marker to be written at the beginning of the table, indicating the number of elements that follow. The length marker is a byte if BYTE or STRING are also given; otherwise the length marker is a WORD. This flag is ignored if LEXV is given

PURE causes the table to be compiled into static memory (ROM).

The flags LENGTH and PURE are implied in LTABLE, PTABLE or PLTABLE.

Examples:

<TABLE 1 2 3 4>    -->

Element 0 WORD	Element 1 WORD	Element 2 WORD	Element 3 WORD
1	2	3	4



<TABLE (BYTE LENGTH) 1 2 3 4> -->

Element 0 BYTE	Element 1 BYTE	Element 2 BYTE	Element 3 BYTE	Element 4 BYTE
4	1	2	3	4

## TELL

<TELL token-commands ...>

Print formatted text to screen. There is a set built-in tokens that can be replaced with TELL-TOKENS or expanded with ADD-TELL-TOKENS.

The built-in tokens are:

Pattern	Form	Description
(CR CRLF)	<CRLF>	Print CR
D *	<PRINTD .X>	Print object-description
N *	<PRINTN .X>	Print number
C *	<PRINTC .X>	Print character
B *	<PRINTB .X>	Print unpacked-string

Example:

```
<TELL "You have " N ,SCORE " points." CR>
--> "You have 42 points.\n"
```

## THROW

<THROW value stack-frame>

**Zapf syntax**

THROW

**Inform syntax**

throw

Versions: 5-

Used in conjunction with CATCH. THROW sets the stack to stack-frame and returns value (the result is that execution returns from the routine where the stack-frame were "caught" with value as the routines return value. Also see CATCH.

Example:

```
<ROUTINE TEST-CATCH ("AUX" X)
  <SET X <CATCH>>
  <THROWER .X>
  123
>

<ROUTINE THROWER (F)
```

```

        <THROW 456 .F>
>

<TEST-CATCH>    -->    456

```

## USL

```
<USL>
```

<b>Zapf syntax</b>	<b>Inform syntax</b>
USL	show_status

Versions: 3

Update status line. In other versions than 3 this command is ignored.

## VALUE

```
<VALUE name/number>
```

<b>Zapf syntax</b>	<b>Inform syntax</b>
VALUE	load

All versions

Load name/number. Command is mostly redundant and rarely used.

Examples:

```

<VALUE X> -->    Loads local or global variable X. Recommended
                  to use LVAL or GVAL instead (.X or ,X)

```

## VERIFY

```
<VERIFY>
```

<b>Zapf syntax</b>	<b>Inform syntax</b>
VERIFY	verify

All versions

Returns true if  $\text{sum}(\$0040:\text{PLENTH (byte 26-27 in header)}) \bmod \$10000 = \text{PCHKSUM (byte 28-29 in header)}$ , otherwise false.

## VERSION?

```
<VERSION? (name/number expressions...)...>
```

VERSION? Lets the game use different logic depending on which version the game is compiled in. The version is read from ZVERSION (byte 0-1) in the header. Valid name/number are:

```

3      ZIP
4      EZIP

```

```

5      XZIP
6      YZIP
7
8
      ELSE/T

```

Example:

```

<VERSION?
    (ZIP <SET X 1> <SET Y 1>)
    (XZIP <SET X 2> <SET Y 2>)
    (ELSE <SET X 3> <SET Y 2>)
>

```

## WINATTR

<WINATTR window-number flags operation>

<b>Zapf syntax</b>	<b>Inform syntax</b>
WINATTR	window_style

Versions: 6-

Change flags for window-number. The flags are:

- Bit 0: Keep text inside margins
- Bit 1: Scroll when reaching bottom
- Bit 2: Copy text to stream 2 (printer)
- Bit 3: Buffer text and word-wrap

The operations are:

- 0: Set to flags
- 1: Set bits supplied (BOR)
- 2: Clear bits supplied
- 3: Reverse bits supplied

## WINGET

<WINGET window-number property>

<b>Zapf syntax</b>	<b>Inform syntax</b>
WINGET	get_wind_prop

Versions: 6-

Reads property on window-number.

## WINPOS

<WINPOS window-number row column>

<b>Zapf syntax</b>	<b>Inform syntax</b>
WINPOS	move_window

Versions: 6-

Move window-number to position row column (pixels). (1, 1) is in the top left corner.

## WINPUT

<WINPUT window-number property value>

### **Zapf syntax**

WINPUT

### **Inform syntax**

put\_wind\_prop

Versions: 6-

Writes value to property window-number.

## WINSIZE

<WINSIZE window-number height width>

### **Zapf syntax**

WINSIZE

### **Inform syntax**

window\_size

Versions: 6-

Changes size on window-number.

## XPUSH

<XPUSH value stack>

### **Zapf syntax**

XPUSH

### **Inform syntax**

push\_stack

Versions: 6-

Push value on stack.

Example:

<GLOBAL MY-STACK <TABLE 1 0 0 0>>

<XPUSH 123 ,MY-STACK> --> MY-STACK <TABLE 2 0 123 0>

## ZWSTR

<ZWSTR src-table length offset dest-table>

### **Zapf syntax**

ZWSTR

### **Inform syntax**

encode\_text

Versions: 5-

Encode length characters starting at offset from ZSCII word zscii-text and stores result

in 6-byte Z-encoded dest-table.

Example:

```
<GLOBAL SRCBUF <TABLE (STRING) "hello">>
<GLOBAL DSTBUF <TABLE 0 0 0>>

<ZWSTR ,SRCBUF 5 1 ,DSTBUF>
<PRINTB ,DSTBUF>          -->  "hello"
```

## Appendix A: Other Z-machine OP-codes

These OP-codes don't have direct ZIL-equivalent (they are used to call routines and control the program counter).

Sources:

*The Z-Machine Standards Document, Graham Nelson*

ZAPF syntax	Inform Syntax	Description (Z specifications 1.0)
CALL1	call_1s	Executes routine() and stores resulting return value.
CALL2	call_2s	Executes routine(arg1) and stores resulting return value.
CALL	call_vs	The only call instruction in Version 3. It calls the routine with 0, 1, 2 or 3 arguments as supplied and stores the resulting return value. (When the address 0 is called as a routine, nothing happens and the return value is false.)
ICALL1	call_1n	Executes routine() and throws away the result.
ICALL2	call_2n	Executes routine(arg1) and throws away the result.
ICALL	call_vn	Like CALL, but throws away the result.
IXCALL	call_vn2	CALL with a variable number (from 0 to 7) of arguments, then throw away the result. This (and call_vs2) uniquely have an extra byte of opcode types to specify the types of arguments 4 to 7. Note that it is legal to use these opcodes with fewer than 4 arguments (in which case the second byte of type information will just be \$FF).
JUMP	jump	Jump (unconditionally) to the given label. (This is not a branch instruction and the operand is a 2-byte signed offset to apply to the program counter.) It is legal for this to jump into a different routine (which should not change the routine call state), although it is considered bad practice to do so and the Txd disassembler is confused by it.
NOOP	nop	Probably the official "no operation" instruction, which, appropriately, was never operated (in any of the Infocom datafiles): it may once have been a breakpoint.
XCALL	call_vs2	Like IXCALL, but stores the resulting value.

## Appendix B – Field-spec for header

The information here is mostly from *The Z-Machine Standards Document*, Graham Nelson and ZILF Source Code. See *The Z-Machine Standards Document* for a more detailed discussion. The field-spec is used in LOWCORE and LOWCORE-TABLE.

### Ordinary header

Field-spec	Byte	Ver	R/W	Description
ZVERSION	0-1	1-	R	Byte 0 Version number
		1-3	-	Byte 1 Flag 1
			R	Bit 1: Status line type: 0=score/turns, 1=hh:mm
			R	Bit 2: Story file split over two discs
			R	Bit 3: Tandy-bit
			R	Bit 4: Status line not available
			R	Bit 5: Screen-splitting available
			R	Bit 6: Is a proportional font the default
		4-	-	*01 Flag 1
			R	Bit 0: Colors available
			R	Bit 1: Picture displaying available
			R	Bit 2: Bold available
			R	Bit 3: Italic available
			R	Bit 4: Monospace (fixed) font available
			R	Bit 5: Sound effects available
			R	Bit 7: Timed keyboard input available
ZORKID/RELEASEID	2-3	1-	R	Release number (word). Note: Traditionally in Infocom only 11 bits are used for release-id (binary and *3777*). That suggests that the higher 5 bits sometime was used or reserved for other information.
ENDLOD	4-5	1-	R	Base of high memory (byte address)
START	6-7	1-5	R	Initial value of program counter (byte address)
		6	R	Packed address of initial "main" routine
VOCAB	8-9	1-	R	Location of dictionary (byte address)
OBJECT	*10-11	1-	R	Location of object table (byte address)
GLOBALS	*12-13	1-	R	Location of global variables table(byte address)
PURBOT	*14-15	1-	R	Base of static memory (byte address)

FLAGS	*16-17	-	-	Flags 2:
		1-	R/W	Bit 0: Set when transcribing is on
		3-	R/W	Bit 1: Set to force printing in monospace font
		6-	R/W	Bit 2: Int sets to request screen redraw, game clears when it complies with this
		5-	R	Bit 3: If set, game wants to use pictures
		3	R	Bit 4: Amigs ver of "The Lurking Horror" sets this probably sound.
		5-	R	Bit 4: If set, game wants to use UNDO
		5-	R	Bit 5: If set, game wants to use mouse
		5-	R	Bit 6: If set, game wants to use colors
		5-	R	Bit 7: If set, game wants to use sound
		6	R	Bit 8: If set, game wants to use menu
SERIAL	18-19	3-	R	Serial number, YY-part
SERI1	20-21	3-	R	Serial number, MM-part
SERI2	22-23	3-	R	Serial number, DD-part
FWORDS	24-25	2-	R	Location of abbreviations table (byte address)
PLENTH	26-27	3-	R	Length of file
PCHKSUM	28-29	3-	R	File checksum
INTWRD	30-31	4-	R	Interpreter number and version
INTID	30	4-	R	Interpreter number
INTVER	31	4-	R	Interpreter version
SCRWRD	32-33	4-	R	Screen width and height
SCRV	32	4-	R	Screen height(lines), 255 = infinite
SCRH	33	4-	R	Screen width (characters)
HWRD	34-35	5-	R	Screen width in units
VWRD	36-37	5-	R	Screen height in units
FWRD	38-39	-	R	Font width and height
	38	5	R	Font width in units (width of '0')
		6-	R	Font height in units
	39	5	R	Font height in units
		6-	R	Font width in units (width of '0')
LMRG / FOFF	40-41	5-	R	Routines offset (divided by 8)

RMRG / SOFF	42-43	5	R	Static strings offset(divided by 8)
CLRWRD	44-45	5-	R	Default background and foreground color
	44	5-	R	Default background color
	45	5-	R	Default foreground color
TCHARS	46-47	5-	R	Address of terminating characters table (bytes)
CRCNT	48-49	5	R/W	???
TWID	48-49	6-	R	Total width in pixels of text sent to output stream 3
CRFUNC /STDREV	50-51	1-	R/W	Standard revision number
CHRSET	52-53	5-	R	Alphabet table address (bytes), or 0 for default
EXTAB	54-55	5-	R	Header extension table address (bytes)

## Extended header

Field-spec	Byte	Ver	R/W	Description
	0-1	-	R	Number of further words in table
MSLOCKX	2-3	5-	R	X-coordinate of mouse after a click
MSLOCY	4-5	5-	R	Y-coordinate of mouse after a click
MSETBL / UNITBL	6-7	5-	R/W	Unicode translation table (optional)
MSEDIR / FLAGS3	8-9	5-	R/W	Flags 3: Bit 0: If set, game wants to use transparency
MSEINV / TRUFGC	10-11	5-	R/W	True default foreground colour
MSEVRB / TRUBGC	12-13	5-	R/W	True default background colour
MSEWRD	14-15	5-	R/W	
BUTTON	16-17	5-	R/W	
JOYSTICK	18-19	5-	R/W	
BSTAT	20-21	5-	R/W	
JSTAT	22-23	5-	R/W	



### ***Appendix C - Reserved constants, globals & locals***

<b>Name</b>	<b>Type</b>	<b>Value</b>	<b>Description</b>
DO-FUNNY-RETURNS?	GLOBAL	<> Versions 3-4 T Versions 5-	
FALSE-VALUE	CONSTANT	0	
FATAL-VALUE	CONSTANT	2	
REDEFINE	LOCAL	<>	