ZIL Reference Guide

Introduction

There are two classe of commands.

The first class is things that only work outside a "routine". These commands is processed during compilation and are a subset of MDL. The order is important and things needs to be declared before (higher up in the file) before they are used.

The second class is things that only work inside "routines". These commands is processed by til Z-machine during runtime. The order these are organized does not matter.

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`

=== MDL SUBRs and FSUBRs ===

(i.e. "things you can use outside a routine")

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

All these commands is 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 ...>
```

Multiply `numbers`.

Example:

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

+ (add)

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

Add `numbers`.

Example:

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

- (subtract)

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

Subtract first `number` by subsequent `numbers`.

Example:

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

/ (divide)

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

Divide first `number` by subsequent `numbers`.

Example:

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

0?

```
<0? value>
```

Predicate. True if `value` is 0 otherwise false.

1?

```
<1? value>
```

Predicate. True if `value` is 1 otherwise false.

==?

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

=?

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

ADD-TELL-TOKENS

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

ADD-WORD

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

ADJ-SYNONYM

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

AGAIN

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

ALLTYPES

```
     <ALLTYPES>
```

returns a `VECTOR` containing just those `ATOM`s which can currently be returned by `TYPE` or `PRIMTYPE`.

AND

```
     <AND conditions ...> **F
```

AND?

```
     <AND? Values ...>
```

ANDB

```
     <ANDB numbers ...>
```

APPLICABLE?

```
     <APPLICABLE? Value>
```

APPLY

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

APPLYTYPE

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

ASCII

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

ASSIGNED?

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

ASSOCIATIONS

```
     <ASSOCIATIONS>
```

ATOM

```
     <ATOM pname>
```

AVALUE

```
     <AVALUE asoc>
```

BACK

```
     <BACK structure [count]>
```

BIND

```
<BIND [activation-atom] (bindings ...)
          [body-decl] body ...> **F
```

BIT-SYNONYM

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

BLOCK

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

BOUND

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

BUZZ

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

BYTE

```
<BYTE number>
```

CHECK-VERSION?

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

CHRSET

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

CHTYPE

```
<CHTYPE value type-atom>
```

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

There is a shortform to change type by typing `#type-atom 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 rest>
```

CONSTANT

```
<CONSTANT atom-or-adecl value> **F
```

CRLF

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

DECL-CHECK

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

DECL?

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

DEFAULT-DEFINITION

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

DEFINE

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

DEFINE-GLOBALS

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

DEFINE20

```
<DEFINE20 name [activation-atom] arg-list [decl] body ...> **F
```

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>
```

END-DEFINITIONS

```
<END-DEFINITIONS>
```

ENDBLOCK

```
<ENDBLOCK>
```

ENDPACKAGE

```
<ENDPACKAGE>
```

ENDSECTION

```
<ENDSECTION>
```

ENTRY

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

EQVB

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

ERROR

```
<ERROR values ...>
```

EVAL

```
<EVAL value [environment]>
```

EVALTYPE

```
<EVALTYPE atom [handler]>
```

EXPAND

```
<EXPAND value>
```

FILE-FLAGS

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

FILE-LENGTH

```
<FILE-LENGTH channel>
```

FLOAD

```
<FLOAD filename>
```

FORM

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

FUNCTION

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

FUNNY-GLOBALS?

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

G=?

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

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

G?

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

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

GASSIGNED?

```
<GASSIGNED? Atom>
```

GBOUND?

```
<GBOUND? Atom>
```

GC

```
<GC>
```

GDECL

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

GET-DECL

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

GETB

```
<GETB table index>
```

GETPROP

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

GLOBAL

```
<GLOBAL atom-or-adecl default-value [decl] [size]> **F
```

GROW

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

GUNASSIGN

```
<GUNASSIGN atom>
```

GVAL

```
<GVAL atom>
```

IFFLAG

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

ILIST

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

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>
```

ISTRING

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

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

ITEM

```
<ITEM asoc>
```

IVECTOR

```
<IVECTOR count [init]>
```

L=?

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

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

L?

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

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

LANGUAGE

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

LEGAL?

```
<LEGAL? Value>
```

LENGTH

```
<LENGTH structure>
```

LENGTH?

```
<LENGTH? structure limit>
```

LINK

```
<LINK value str oblist>
```

LIST

```
<LIST values ...>
```

LONG-WORDS?

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

LOOKUP

```
<LOOKUP str oblist>
```

LPARSE

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

LSH

```
<LSH number1 number2>
```

LTABLE

```
<LTABLE [flag-list] values ...>
```

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

LVAL

```
<LVAL atom [environment]>
```

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 ...>
```

MEMBER

```
<MEMBER item structure>
```

MEMQ

```
<MEMQ item structure>
```

MIN

```
<MIN numbers ...>
```

MOBLIST

```
<MOBLIST name>
```

MOD

```
<MOD number1 number2>
```

MSETG

```
<MSETG atom-or-adecl value> **F
```

N==?

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

N=?

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

NEW-ADD-WORD

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

NEWTYPE

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

NEXT

```
<NEXT asoc>
```

NOT

```
<NOT value>
```

NTH

```
<NTH structure index>
```

OBJECT

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

OBLIST?

```
<OBLIST? Atom>
```

OFFSET

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

OPEN

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

OR

```
<OR conditions ...> **F
```

OR?

```
<OR? Values ...>
```

ORB

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

ORDER-FLAGS?

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

ORDER-OBJECTS?

```
<ORDER-OBJECTS? Atom>
```

ORDER-TREE?

```
<ORDER-TREE? Atom>
```

PACKAGE

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

PARSE

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

PLTABLE

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

Defines a table containing the specified `values` and with the `LENGTH` and `PURE` flag (see `TABLE`).

PNAME

```
<PNAME atom>
```

PREP-SYNONYM

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

PRIMTYPE

```
<PRIMTYPE value>
```

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]>
```

PRINC

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

PRINT

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

PRINT-MANY

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

PRINTTYPE

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

PROG

```
<PROG [activation-atom] (bindings ...)
          [body-decl] body ...> **F
```

PROPDEF

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

PTABLE

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

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

PUT

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

PUT-DECL

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

PUTB

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

PUTPROP

```
<PUTPROP item indicator [value]>
```

PUTREST

```
<PUTREST list new-rest>
```

QUIT

```
<QUIT [exit-code]>
```

QUOTE

```
<QUOTE value> **F
```

READSTRING

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

REMOVE

```
<REMOVE {atom | pname oblist}>
```

RENTRY

```
<RENTRY atoms ...>
```

REPEAT

```
<REPEAT [activation-atom] (bindings ...)
          [body-decl] body ...> **F
```

REPLACE-DEFINITION

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

REST

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

RETURN

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

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]>
```

SET-DEFSTRUCT-FILE-DEFAULTS

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

SETG

```
<SETG atom value>
```

SETG20

```
<SETG20 atom value>
```

SORT

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

SPNAME

```
<SPNAME atom>
```

STRING

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

STRUCTURED?

```
<STRUCTURED? Value>
```

SUBSTRUC

```
<SUBSTRUC structure [rest] [amount] [structure]>
```

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 ...>
```

Defines a table containing the specified `values`.

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).

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-TOKENS {pattern form} ...> **F
<TOP structure>
```

```
<TUPLE values ...>
```

TYPE

```
<TYPE value>
```

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>
```

Evaluates to type-i only if `<==? type-i >` is true. It is faster and gives more information than `OR`ing 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>
```

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
<PRIMTYPE BYTE>
--> FIX
```

UNASSIGN

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

UNPARSE

```
<UNPARSE value>
```

USE

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

USE-WHEN

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

VALID-TYPE?

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

VALUE

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

VECTOR

```
<VECTOR values ...>
```

VERB-SYNONYM

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

VERSION

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

VERSION?

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

VOC

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

XORB

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

ZGET

```
<ZGET table index>
```

ZIP-OPTIONS

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

ZPUT

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

ZREST

```
<ZREST table bytes>
```

ZSTART

```
<ZSTART atom>
```

=== *Z-code builtins* ===

(i.e. "things you can use inside a routine")

Sources:

The Z-Machine Standards Document, Graham Nelson

The Inform Designer's Manual, Graham Nelson

ZIL Language Guide, Jesse McGrew

* (multiply)

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

Zapf syntax	**Inform syntax**
MUL	mul

Multiply `numbers`.

Example:

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

+ (add)

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

Zapf syntax	**Inform syntax**
ADD	add

Add `numbers`.

Example:

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

- (subtract)

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

Zapf syntax	**Inform syntax**
SUB	sub

Subtract first `number` by subsequent `numbers`.

Example:

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

/ (divide)

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

Zapf syntax	**Inform syntax**
DIV	div

Divide first `number` by subsequent `numbers`.

Example:

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

0?

```
     <0? value>
```

Zapf syntax	**Inform syntax**
ZERO?	Jz

Predicate. True if `value` is 0 otherwise false.

1?

```
     <1? value>
```

Predicate. True if `value` is 1 otherwise false.

==?, =?, EQUAL?

```
     <==? value values...>
     <=? value values...>
```

Zapf syntax	**Inform syntax**
EQUAL?	Je

Both predicates evaluates to the same Zapf syntax and are synonyms.

AGAIN

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

Skips the rest of the loop and starts again from the top.

AND

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

Logical AND.

APPLY

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

ASH

```
     <ASH number places>
```

Zapf syntax	**Inform syntax**
ASHIFT	art_shift

ASSIGNED?

```
     <ASSIGNED? Name>
```

Zapf syntax	Inform syntax
ASSIGNED?	check_arg_count

BACK

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

BAND

```
<BAND numbers...>
```

Zapf syntax	Inform syntax
BAND	and

Bitwise OR.

BCOM

```
<BCOM value>
```

Zapf syntax	Inform syntax
BCOM	not

BIND

```
<BIND (bindings...) expressions...>
```

BOR

```
<BOR numbers...>
```

Zapf syntax	Inform syntax
BOR	or

Bitwise OR.

BUFOUT

```
<BUFOUT value>
```

Zapf syntax	Inform syntax
BUFOUT	buffer_mode

CATCH

```
<CATCH>
```

Zapf syntax	Inform syntax
CATCH	catch

CHECKU

```
<CHECKU character>
```

Zapf syntax	Inform syntax
CHECKU	check_unicode

CLEAR

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

Zapf syntax	Inform syntax
CLEAR	erase_window

COLOR

```
<COLOR fg bg>
```

Zapf syntax	Inform syntax
COLOR	set_colour

COND

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

COPYT

```
<COPYT src-table dest-table length>
```

Zapf syntax	Inform syntax
COPYT	copy_table

CRLF

```
<CRLF>
```

Zapf syntax	Inform syntax
CRLF	new_line

CURGET

```
<CURGET table>
```

Zapf syntax	Inform syntax
CURGET	get_cursor

CURGET is only available in version 4 and later.

CURSET

```
<CURSET row column>
```

CURSET is only available in version 4 and later.

DCLEAR

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

Zapf syntax	Inform syntax
DCLEAR	erase_picture

DEC

```
<DEC name>
```

Zapf syntax	Inform syntax
DEC	dec

DIRIN

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

Zapf syntax	Inform syntax
DIRIN	input_stream

DIROUT

```
<DIROUT stream-number [table] [width]>
```

Zapf syntax	Inform syntax
DIROUT	output_stream

DISPLAY

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

Zapf syntax	Inform syntax
DISPLAY	draw_picture

DLESS?

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

Zapf syntax	Inform syntax
DLESS?	dec_chk

DO

```
<DO (name start end [step]) 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	Inform syntax
ERASE	erase_line

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 the 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.

F?

```
<F? expression>
```

FCLEAR

```
<FCLEAR object flag>
```

Zapf syntax	Inform syntax
FCLEAR	clear_attr

FIRST?

```
<FIRST? Object>
```

Zapf syntax	Inform syntax
FIRST?	get_child

FONT

```
<FONT number>
```

Zapf syntax	Inform syntax
FONT	set_font

FSET

```
<FSET object flag>
```

Zapf syntax	Inform syntax

FSET	set_attr

FSET?

```
<FSET? object flag>
```

Zapf syntax	**Inform syntax**
FSET?	test_attr

FSTACK

```
<FSTACK [stack]>
```

Zapf syntax	**Inform syntax**
FSTACK	pop / pop_stack

G?

```
<G? value value>
```

Zapf syntax	**Inform syntax**
GRTR?	Jg

G=?

```
<G=? value value>
```

GET

```
<GET table offset>
```

Zapf syntax	**Inform syntax**
GET	loadw

GETB

```
<GETB table offset>
```

Zapf syntax	**Inform syntax**
GETB	loadb

GETP

```
<GETP object property>
```

Zapf syntax	**Inform syntax**
GETP	get_prop

GETPT

```
<GETPT object property>
```

Zapf syntax	**Inform syntax**
GETPT	get_prop_addr

GVAL

```
<GVAL name>
```

HLIGHT

```
<HLIGHT style>
```

Zapf syntax	Inform syntax
HLIGHT	set_text_style

0	Normal
1	Inverse
2	Bold
4	Italic
8	Mono

`HLIGHT` is only available in version 4 and later.

IFFLAG

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

IGRTR?

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

Zapf syntax	Inform syntax
IGRTR?	inc_chk

Increment `name`, and test if `name` is greater than `value`.

IN?

```
<IN? object object>
```

Zapf syntax	Inform syntax
IN?	jin

INC

```
<INC name>
```

Zapf syntax	Inform syntax
INC	inc

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

INPUT

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

Zapf syntax	Inform syntax

INPUT	read_char

`INPUT` is only available in version 4 and later. `INPUT` reads a single character from the keyboard.

An example of using the optional arguments time and routine. This creates a pause of two seconds (if not interrupted by a keypress from the player):

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

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

INTBL?

```
<INTBL? value table length [form]>
```

Zapf syntax	**Inform syntax**
INTBL?	scan_table

IRESTORE

```
<IRESTORE>
```

Zapf syntax	**Inform syntax**
IRESTORE	restore_undo

ISAVE

```
<ISAVE>
```

Zapf syntax	**Inform syntax**
ISAVE	save_undo

ITABLE

```
<ITABLE [length-spec] number [(table-flags...)]
          [const-expressions...]>
```

L?

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

Is `value1` lower to `value2`.

Zapf syntax	**Inform syntax**
LESS?	Jl

L=?

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

Is `value1` lower or equal to `value2`.

LEX

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

Zapf syntax	Inform syntax
LEX	tokenise

LEX is only available in version 4 and later. Parse the text into parse. See READ for more info about parsing.

LOC

```
<LOC object>
```

Zapf syntax	Inform syntax
LOC	get_parent

LOWCORE-TABLE

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

LOWCORE

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

LSH

```
<LSH number places>
```

Zapf syntax	Inform syntax
SHIFT	log_shift

LTABLE

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

LVAL

```
<LVAL name>
```

MAP-CONTENTS

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

MAP-DIRECTIONS

```
<MAP-DIRECTIONS (name pt room) expressions...>
```

MARGIN

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

Zapf syntax	Inform syntax
MARGIN	set_margins

MENU

```
<MENU number table>
```

Zapf syntax	Inform syntax
MENU	make_menu

MOD

```
<MOD number number>
```

Zapf syntax	Inform syntax
MOD	mod

MOUSE-INFO

```
<MOUSE-INFO table>
```

Zapf syntax	Inform syntax
MOUSE-INFO	read_mouse

MOUSE-LIMIT

```
<MOUSE-LIMIT window-number>
```

Zapf syntax	Inform syntax
MOUSE-LIMIT	mouse_window

MOVE

```
<MOVE object object>
```

Zapf syntax	Inform syntax
MOVE	insert_obj

N=?

```
<N==? value values...>
```

NEXT?

```
<NEXT? Object>
```

Zapf syntax	Inform syntax
NEXT?	get_sibling

NEXTP

```
<NEXTP object property>
```

Zapf syntax	Inform syntax
NEXTP	get_next_prop

NOT

```
<NOT expression>
```

Logical NOT.

OR

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

Logical AND.

ORIGINAL

```
<ORIGINAL?>
```

Zapf syntax	**Inform syntax**
`ORIGINAL?`	`piracy`

PICINF

```
<PICINF picture-number table>
```

Zapf syntax	**Inform syntax**
`PICINF`	`picture_data`

PICSET

```
<PICSET table>
```

Zapf syntax	**Inform syntax**
`PICSET`	`picture_table`

PLTABLE

```
<PLTABLE [(table-flags...)] values...>
```

POP

```
<POP [stack]>
```

Zapf syntax	**Inform syntax**
`POP`	`pull`

PRINT

```
<PRINT packed-string>
```

Zapf syntax	**Inform syntax**
`PRINT`	`print_paddr`

PRINTB

```
<PRINTB unpacked-string>
```

Zapf syntax	**Inform syntax**
`PRINTB`	`print_addr`

PRINTC

```
<PRINTC character>
```

Zapf syntax	**Inform syntax**

PRINTC	print_char

PRINTD

```
<PRINTD object>
```

Zapf syntax	**Inform syntax**
PRINTD	print_obj

PRINTF

```
<PRINTF table>
```

Zapf syntax	**Inform syntax**
PRINTF	print_form

PRINTI

```
<PRINTI string>
```

Zapf syntax	**Inform syntax**
PRINTI	print

PRINTN

```
<PRINTN number>
```

Zapf syntax	**Inform syntax**
PRINTN	print_num

PRINTR

```
<PRINTR string>
```

Zapf syntax	**Inform syntax**
PRINTR	print_ret

PRINTT

```
<PRINTT table width height skip>
```

Zapf syntax	**Inform syntax**
PRINTT	print_table

PRINTU

```
<PRINTU number>
```

Zapf syntax	**Inform syntax**
PRINTU	print_unicode

PROG

```
<PROG (bindings...) expressions...>
```

PTABLE

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

PTSIZE

```
<PTSIZE table>
```

Zapf syntax	**Inform syntax**
PTSIZE	get_prop_len

PUSH

```
<PUSH value>
```

Zapf syntax	**Inform syntax**
PUSH	push

PUT

```
<PUT table offset value>
```

Zapf syntax	**Inform syntax**
PUT	storew

PUTB

```
<PUTB table offset value>
```

Zapf syntax	**Inform syntax**
PUTB	storeb

PUTP

```
<PUTP object property value>
```

Zapf syntax	**Inform syntax**
PUTP	put_prop

QUIT

```
<QUIT>
```

Zapf syntax	**Inform syntax**
QUIT	quit

RANDOM

```
<RANDOM range>
```

Zapf syntax	**Inform syntax**
RANDOM	random

READ

```
<READ text parse [time] [routine]>
```

Zapf syntax	**Inform syntax**
READ	aread / sread

Reads text from keyboard and parse it. Result is stored in two byte-tables. Byte 0 in `text` most contain the max-size of the buffer and if `parse` is supplied, byte 0 of it most cointain 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	Adress to first word in dictionary (0 if word is not in it)
	4	Length of first word
	5	Start position (in `text`) 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 info about `READ`.

REMOVE

```
<REMOVE object>
```

Zapf syntax	**Inform syntax**
REMOVE	remove_obj

REPEAT

```
<REPEAT (bindings...) expressions...>
```

REST

```
<REST table [bytes]>
```

RESTART

```
<RESTART>
```

Zapf syntax	Inform syntax
RESTART	restart

RESTORE

```
<RESTORE [table] [bytes] [filename]>
```

Zapf syntax	Inform syntax
RESTORE	restore

RETURN

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

Zapf syntax	Inform syntax
RETURN	ret

RFALSE

```
<RFALSE>
```

Zapf syntax	Inform syntax
RFALSE	rfalse

RFATAL

```
<RFATAL>
```

RSTACK

```
<RSTACK>
```

Zapf syntax	Inform syntax
RSTACK	ret_popped

RFALSE

```
<RTRUE>
```

Zapf syntax	Inform syntax
RTRUE	rtrue

SAVE

```
<SAVE [table] [bytes] [filename]>
```

Zapf syntax	**Inform syntax**
SAVE	save

SCREEN

```
<SCREEN window-number>
```

Zapf syntax	**Inform syntax**
SCREEN	set_window

SCROLL

```
<SCROLL window-number pixels>
```

Zapf syntax	**Inform syntax**
SCROLL	scroll_window

SET

```
<SET name value>
```

Zapf syntax	**Inform syntax**
SET	store

SETG

```
<SETG name value>
```

SOUND

```
<SOUND number [effect] [volume] [routine]>
```

Zapf syntax	**Inform syntax**
SOUND	sound_effect

SPLIT

```
<SPLIT number>
```

Zapf syntax	**Inform syntax**
SPLIT	split_window

T?

```
<T? expression>
```

TABLE

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

TELL

```
<TELL token-commands>
```

THROW

```
<THROW value stack-frame>
```

Zapf syntax	Inform syntax
THROW	throw

USL

```
<USL>
```

Zapf syntax	Inform syntax
USL	show_status

VALUE

```
<VALUE name/number>
```

Zapf syntax	Inform syntax
VALUE	load

VERIFY

```
<VERIFY>
```

Zapf syntax	Inform syntax
VERIFY	verify

VERSION?

```
<VERSION? (name/number expressions...)...>
```

WINATTR

```
<WINATTR window-number flags operation>
```

Zapf syntax	Inform syntax
WINATTR	window_style

WINGET

```
<WINGET window-number property>
```

Zapf syntax	Inform syntax
WINGET	get_wind_prop

WINPOS

```
<WINPOS window-number row column>
```

Zapf syntax	Inform syntax
WINPOS	move_window

WINPUT

```
<WINPUT window-number property value>
```

Zapf syntax	Inform syntax
WINPUT	put_wind_prop

WINSIZE

```
<WINSIZE window-number height width>
```

Zapf syntax **Inform syntax**

```
WINSIZE            window_size
```

XPUSH

```
<XPUSH value stack>
```

Zapf syntax **Inform syntax**

```
XPUSH              push_stack
```

ZWSTR

```
<ZWSTR src-table length offset dest-table>
```

Zapf syntax **Inform syntax**

```
ZWSTR              encode_text
```

=== Other Z-machine OP-codes ===

These OP-codes don't have direct ZIL-equivalent (used to call routines).

Sources:

The Z-Machine Standards Document, Graham Nelson

ZAPF syntax	**Inform Syntax**	**Description (Z specifikations 1.0)**
BTST	test	Jump if all of the flags in bitmap are set (i.e. if bitmap & flags == flags).
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 result.
ICALL2	call_2n	Executes routine(arg1) and throws away result.
ICALL	call_vn	Like CALL, but throws away 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 resulting value.