ZILF Reference Guide

Henrik Åsman et al.

Copyright (C) 2020 Henrik Åsman

Copying and distribution of this file, with or without modification, are permitted in any medium without royalty provided the copyright notice and this notice are preserved. This file is offered as-is, without any warranty.

Table of Contents

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: the game we now call 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 for Interactive Fiction.

Infocom's development environment was always 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, although 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 underway 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: the compiler known as 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

Goal of document

[*This NOT a manual on how to write games. The goal is to list all instructions and syntaxes with short examples to use when reading game code. It should also list syntax and behaviour that is specific for ZILF.*]

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"
```

Regarding ATOMs and other primitive types

ZILF recognizes the following primitive types: ATOM, FIX, LIST, STRING, TABLE and VECTOR. Everything that ZILF encounters when it reads the code falls into one of these types (or derived types of these primitive ones).

FIX	32-bit signed integer. Any number is a FIX. CHARACTER and BYTE are examples of TYPEs whose PRIMTYPE is FIX.	616 *747* #2 10110111 !\A
LIST	Linked list (forward looking).A LIST is enclosed by parentheses. FORM, FUNCTION and MACRO are examples of TYPEs whose PRIMTYPE is LIST.	(1 2 3) <+ 1 2> <> #FUNCTION <() <+ 1 2>>
STRING	A continuous byte array containing	"Hello, world!"

	characters. A STRING is enclosed by double-quotes.	
TABLE	A continuous byte or word array. This TYPE is specific for ZIL and is a simplified VECTOR without any TYPE information about the individual elements. A TABLE is zero-based and element at index 0 can be specified to contain the length of the TABLE.	#TABLE [5 6 7] <TABLE (BYTE) 5 6 7> <TABLE (BYTE LENGTH) 5 6 7>
VECTOR	A continuous array of elements. A VECTOR is enclosed by square brackets. Each element holds information about its TYPE. MDL has a UVECTOR TYPE that contains uniform elements (same TYPE) but ZILF handles these as VECTORs.	[1 !\A "Hello" (1 2 3)] ![1 2 3 4 5!]

Everything that does not have one of the above as its PRIMTYPE is an ATOM. One can think of an ATOM as a variable that can hold one of the other TYPEs. Every ATOM can have a global value and a local value. Every ATOM also has a PNAME ("print name") that ZILF uses when it needs to print the name of the ATOM.

Examples:

```
;"Assign the global ATOM BAR the FIX 123"
<SETG BAR 123>

;"Assign the local ATOM BAR the VECTOR [1 2 3]"
<SET BAR [1 2 3]>

;"Assign the global ATOM BARBAR the global value of ATOM BAR"
<SETG BARBAR <GVAL BAR>>
,BARBAR                         -->  123

;"Assign the local ATOM FOOBAR the ATOM BAR"
<SET FOOBAR BAR>
.FOOBAR                         -->  BAR
<LVAL <LVAL FOOBAR>>            -->  [1 2 3]
,.FOOBAR                        -->  123

;"Assign the global ATOM FUNC a FUNCTION"
<DEFINE FUNC () <+ 1 2>>

;"Assign the global ATOM BAZ a FORM"
<SETG BAZ '<+ 1 2>>
,BAZ                            -->  <+ 1 2>
<EVAL ,BAZ>                     -->  3
```

DECL and ADECL

Regarding OBLISTs

[*Short text explaining how OBLISTs are used for "Lexical Blocking", mostly from The MDL Programming Language, chapter 15. All references in the document to chapter 15 should refer to here instead.*]

% and %%

When ZILF interprets MDL, either during compilation or when using ZILF in interpreter mode, it goes through three stages repeatedly, `READ`, `EVAL` and `PRINT`.

READ reads ASCII-text and when it has something enclosed in matching brackets the result is passed to EVAL for evaluation that in its turn passes the result of the evaluation to PRINT that prints the evaluated result. The flow is like below:

printable text → `READ` → [MDL objects] → `EVAL` → [MDL objects] → `PRINT` → printable text

Consider a simple statement like: `<+ 1 2>`

`READ` reads from left to right and when it encounters the closing bracket the MDL object `<+ 1 2>`, a `FORM`, is passed to `EVAL`. `EVAL` evaluates this MDL object and the resulting MDL object, 3, is passed to `PRINT` for printing.

`%` and `%%` are "`READ` macros" that are used to modify this process. Whenever `READ` encounters `%` or `%%` it immediately passes the MDL object that follows the `%`, or `%%`, to `EVAL` before it continues the `READ`.

In case of `%` the result of the `EVAL` is inserted in place of the MDL object that follows the `%` and is used by the following `READ`.

In case of `%%` the result of the `EVAL` is ignored and nothing is inserted in place of the MDL object that follows the `%%` (eventual side effects of the `EVAL` remains, of course).

Example:

```
<DEFINE INITA () <SETG A 0>>
<DEFINE INCA () <SETG A <+ ,A 1>>>

;"1st INIT-A, 1st INC-A, 2nd INC-A, 2nd INIT-A, 3rd INC-A"
[<INIT-A> <INC-A> <INC-A> (<INIT-A>) <INC-A>]
                                        -->  [0 1 2 (0) 1]

;"1st INIT-A, 3rd INC-A, 1st INC-A, 2nd INC-A, 2nd INIT-A"
[%<INIT-A> <INC-A> <INC-A> (<INIT-A>) %<INC-A>]
                                        -->  [0 2 3 (0) 1]

;"2nd INIT-A, 3rd INC-A, 1st INIT-A, 2nd INC-A, 3rd INC-A"
[<INIT-A> <INC-A> <INC-A> (%<INIT-A>) %<INC-A>]
                                        -->  [0 1 2 (0) 1]

;"1st INIT-A, 1st INC-A, 2nd INC-A, 2nd INIT-A, 3rd INC-A"
[%%<INIT-A> %<INC-A> %<INC-A> (%%<INIT-A>) %<INC-A>]
                                        -->  [1 2 () 1]
```

Segments

Segments are used to copy elements from one structure TYPE to another structure TYPE. Segments take the form !<function args …!> where the second exclamation point is optional. The implicit form of LVAL and GVAL is legal. The segment is EVALuated and must be EVALuated inside another structure and the result of the EVAL must be a structure, otherwise an error is raised.

Examples:

```
<SET L0 [4 5]>
<SET L1 (1 2 3 .L0)>            -->  (1 2 3 [4 5])
<SET L2 [!<LVAL L1!>]>          -->  [1 2 3 [4 5]]
<1 .L0 6>
.L1                             -->  (1 2 3 [6 5])
.L2                             -->  [1 2 3 [6 5]]
[!<SUBSTRUC .L1 0 3> (!.L0)]    -->  [1 2 3 (6 5)]
```

What is the "new parser"?

[Explain what type of animal the new parser is.]

MDL built-ins 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:

`MDL built-in`	MDL built-in function. Part of MUDDLE.56 on ITS. *The MDL Programming Language,* *S. W. Galley and Greg Pfister* *MUDDLE F/SUBRS (MUDMAN for MUDDLE 55),* *P. David Lebling and S. W. Galley*
`MDL package system`	Support for lexical blocking. *The MDL Programming Environment, P. David Lebling*
`ZIL library`	Functions added through ZIL/ZILCH at Infocom to support building of interactive fiction. *ZIL Language Guide, Jesse McGrew* *ZILF source code and test cases, Jesse McGrew* *Learning ZIL, Steven E. Meretzky* *ZIL, Marc S. Blank*
`ZILF compiler directive`	*ZILF source code and test cases, Jesse McGrew*

* (multiply)

```
<* numbers ...>

MDL built-in
```

Multiply `numbers`.

Example:

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

+ (add)

```
<+ numbers ...>

MDL built-in
```

Add `numbers`.

Example:

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

- (subtract)

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

```
MDL built-in
```

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 built-in
```

Divide first `number` by subsequent `numbers`.

Example:

```
</ 20 5 2>     -->  2
```

0?

```
<0? value>

MDL built-in
```

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

1?

```
<1? value>

MDL built-in
```

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

==?

```
<==? value1 value2>

MDL built-in
```

This is a predicate that returns `TRUE` if `value1` and `value2` is the same object, otherwise it returns `FALSE`.

For `ATOM`s whose `TYPE` are structured (for example `LIST`s and `VECTOR`s) the `ATOM`s must refer (point) to the same structure to be considered ==. These `ATOM`s are actually pointers that point to an memory address and the two `ATOM`s must point to the same address to be ==.

For `ATOM`s whose `TYPE` is not structured the `ATOM`s are considered == if they are of the same `TYPE` and contain the same value.

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

- 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 (in ZILF, but not in MDL)
```

=?

```
<=? value1 value2>

MDL built-in
```

This is a predicate that returns TRUE if value1 and value2 is of the same TYPE and structurally equal, otherwise it returns 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 (zillib 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]>

ZIL parser library
```

ADD-WORD requires the new parser (<SETG NEW-PARSER? T>). Note that the standard library that's included with ZILF, zillib, doesn't support the new parser.

The new parser needs a couple of boot-strap FUNCTIONs, GVALs and DEFSTRUCTs to work.

CLASSIFIED	A global LIST that defines the part-of-speech and its value.
GET-CLASSIFICATION	A FUNCTION that can return the part-of-speech from CLASSIFIED.
VERB-DATA	A DEFSTRUCT.
VWORD	A DEFSTRUCT.

There also needs to be a call to SET-DEFSTRUCT-FILE-DEFAULTS to set up the DEFSTRUCTs.

There's also two COMPILATION-FLAGs that control how the vocabulary is set up.

WORD-FLAGS-IN-TABLE	Creates the GVAL WORD-FLAG-TABLE.
ONE-BYTE-PARTS-OF-SPEECH	Control if the part-of-speech value should occupy a byte or a word (If the size of each entry in the vocabulary is 9 or 10 bytes)

INFOCOM only used the new parser in three published games (Arthur, Shogun and Zork Zero) and two unpublished projects (Abyss and Restaurant). ADD-WORD and NEW-ADD-WORD is in these games called with these values

part-of-speech	value	flags	flag-value
ADJ	<>	FIRST-PERSON	8
ADV	<VOC string>	PLURAL-FLAG	16
APOSTR		SECOND-PERSON	32
ARTICLE		THIRD-PERSON	64
ASKWORD		PRESENT-TENSE	256
CANDO		PAST-TENSE	512
COMMA		FUTURE-TENSE	1024
END-OF-INPUT		POSSESSIVE	16384
MISCWORD			
NOUN			
OFWORD			
PARTICLE			
PREP			
QUANT			
QUOTE			
QWORD			
TOBE			
VERB			

Examples:

```
<VERSION 5>

<COMPILATION-FLAG WORD-FLAGS-IN-TABLE T>
<COMPILATION-FLAG ONE-BYTE-PARTS-OF-SPEECH <>>

<SETG NEW-PARSER? T>

<SETG CLASSIFICATIONS '(ADJ 1 BUZZ 2 DIR 4 NOUN 8 PREP 16
                        VERB 32 PARTICLE 64 ARTICLE 128
                        ASKWORD 256 QUOTE 512)>

<DEFINE GET-CLASSIFICATION (TYPE "AUX" P)
    <COND (<SET P <MEMQ .TYPE ,CLASSIFICATIONS>> <2 .P>)
          (T <ERROR NO-SUCH-WORD-TYPE!-ERRORS>)>>

<SET-DEFSTRUCT-FILE-DEFAULTS ('START-OFFSET 0)
                             ('PUT ZPUT)
                             ('NTH ZGET)>

<DEFSTRUCT VERB-DATA (TABLE ('INIT-ARGS (TEMP-TABLE)))
    (VERB-ZERO ANY -1)
    (VERB-RESERVED FALSE)
    (VERB-ONE <OR FALSE TABLE>)
    (VERB-TWO <OR FALSE TABLE>)>

<DEFSTRUCT VWORD (TABLE ('INIT-ARGS (TEMP-TABLE)))
    (WORD-LEXICAL-WORD ANY)
    (WORD-CLASSIFICATION-NUMBER FIX)
    (WORD-FLAGS FIX)
    (WORD-SEMANTIC-STUFF ANY)
    (WORD-VERB-STUFF ANY)
```

```
        (WORD-ADJ-ID ANY)
        (WORD-DIR-ID ANY)>

    <SYNTAX SING = V-SING>
    <ROUTINE V-SING () <>>

    <SYNTAX ATTACK OBJECT WITH OBJECT = V-ATTACK>
    <ROUTINE V-ATTACK () <>>

    <ADD-WORD FOO NOUN <> 12345>
    <ADD-WORD BAR PREP>
    <SYNONYM BAR BAZ>

    <ROUTINE GO () <TEST-NEW-PARSER>>

    <ROUTINE TEST-NEW-PARSER ("AUX" S)
         ;"Should affect VOCAB word size"
         <SET S <GETB ,VOCAB <+ 1 <GETB ,VOCAB 0>>>>
         <TELL "VOCAB word-size = " N .S CR>

         ;"Verbs should have verb data"
         <TELL "Verb data = " N <GET ,W?SING 3> CR>

         ;"Should affect syntax format"
         <TELL "Verb WORD 3 (Byte 6-7) in VOCAB is pointer:" CR>
         <TELL "   WORD 0 = " N <GET <GET ,W?ATTACK 3> 0> CR>
         <TELL "   WORD 1 = " N <GET <GET ,W?ATTACK 3> 1> CR>
         <TELL "   WORD 2 = " N <GET <GET ,W?ATTACK 3> 2> CR>
         <TELL "   WORD 3 = " N <GET <GET ,W?ATTACK 3> 3> CR>

         ;"WORD-FLAG-TABLE should list words and flags"
         <TELL "WORD-FLAG-TABLE contain words and flags" CR>
         <TELL "   Entry size = " N <GET ,WORD-FLAG-TABLE 0> CR>
         <TELL "   W?FOO = " N ,W?FOO CR>
         <TELL "   Word = " N <GET ,WORD-FLAG-TABLE 1> CR>
         <TELL "   Flag = " N <GET ,WORD-FLAG-TABLE 2> CR>

         ;"Synonyms should use pointers, part-of-speech = 0"
         <TELL "SYNONYM points to parent" CR>
         <TELL "   W?BAR = " N ,W?BAR CR>
         <TELL "   W?BAZ = " N ,W?BAR CR>
         <TELL "   WORD 3 in W?BAR = " N <GET ,W?BAR 3> CR>
         <TELL "   WORD 3 in W?BAZ = " N <GET ,W?BAZ 3> CR>
         <TELL "   WORD 4 (PoS) in W?BAR = " N <GET ,W?BAR 4> CR>
         <TELL "   WORD 4 (PoS) in W?BAZ = " N <GET ,W?BAZ 4> CR>
    >
```

ADJ-SYNONYM

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

    ZIL parser library
```

`ADJ-SYNONYM` creates one or more `synonyms` to the `original` adjective.

ZILF treats ADJ-SYNONYM as an alias to SYNONYM.

Note that due to the way words, especially adjectives and nouns, are stored in the vocabulary synonyms for adjectives only work in version 3 (ZIP) games.

AGAIN

```
<AGAIN [activation]>

MDL built-in
```

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 built-in
```

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

AND

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

```
MDL built-in
```

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 built-in
```

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 built-in
```

Bitwise AND.

Examples:

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

APPLICABLE?

```
<APPLICABLE? value>

MDL built-in
```

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

Applicable TYPEs:

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

Example:

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

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

APPLY

```
<APPLY applicable args ...>

MDL built-in
```

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]>

MDL built-in
```

APPLYTYPE tells the TYPE atom how it should be applied in a FORM. If APPLYTYPE is called without a handler then the currently active handler is returned. If there is no active handler, FALSE is returned.

Note that it is possible to replace the handler with a new handler, even on the predefined TYPEs (see EVALTYPE for example on this).

See EVALTYPE, NEWTYPE and PRINTTYPE.

Example:

```
<NEWTYPE WINNER LIST>
<APPLYTYPE WINNER>                      -->  #FALSE
```

```
<APPLYTYPE WINNER <FUNCTION (W "TUPLE" T) (!.W !.T)>>
<#WINNER (A B C) <+ 1 2> q>              -->  (A B C 3 q)
```

ASCII

```
<ASCII {number | character}>

MDL built-in
```

Converts `number` to `character` or `character` to `number`.

Examples:

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

ASK-FOR-PICTURE-FILE?

```
<ASK-FOR-PICTURE-FILE?>

ZIL library
```

ZILF ignores this and always returns `FALSE`.

ASSIGNED?

```
<ASSIGNED? atom [environment]>

MDL built-in
```

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>

MDL built-in
```

`ASSOCIATIONS` gives access to the association chain. `ASSOCIATIONS` returns the first entry in the chain or FALSE if the chain is empty. Each entry is of the `TYPE` `ASOC`. An `ASOC` contains three elements: `ITEM`, `INDICATOR` and `VALUE`. An `ASOC` looks like a `LIST` but behaves differently.

Note that ZILF adds new `ASSOCIATIONS` last in the chain instead of at the top that's usually done in MDL.

See `AVALUE`, `GETPROP`, `INDICATOR`, `ITEM`, `NEXT` and `PUTPROP` on how to extract, create and

traverse the chain.

Example:

```
;"Put all ASOCs in a LIST"
<PROG ((A <ASSOCIATIONS>))
      <COND (<NOT .A> '())
            (T (.A !<MAPF ,LIST
                  <FUNCTION () <COND (<SET A <NEXT .A>> .A)
                                     (T <MAPSTOP>)>>>))>>
```

ATOM

```
<ATOM pname>

MDL built-in
```

ATOM returns a newly created ATOM with pname (string). The ATOM is not on any OBLIST and therefore has the trailer !-#FALSE () attached to it.

See *The MDL Programming Language, chapter 15*.

Examples:

```
<ATOM "FOO">                          -->  FOO!-#FALSE ()
<==? <ATOM "FOO"> <ATOM "FOO">>       -->  #FALSE
```

AVALUE

```
<AVALUE asoc>

MDL built-in
```

AVALUE returns the value part from an asoc entry, of TYPE ASOC, in the ASSOCIATION chain.

See ASSOCIATIONS, GETPROP, INDICATOR, ITEM, NEXT and PUTPROP.

Example:

```
<DEFINE LAST-ASOC ()
     <REPEAT ((A <ASSOCIATIONS>))
          <COND (<=? .A <>> <RETURN <>>)
          (<=? <NEXT .A> <>> <RETURN .A>)>
     <SET A <NEXT .A>>>>

<PUTPROP NEW-ASOC TEXT "Hello, world!">
<SET A <LAST-ASOC>>
<AVALUE .A>                    -->  "Hello, world!"
```

BACK

```
<BACK array [count]>

MDL built-in
```

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]
```

BEGIN-SEGMENT

```
<BEGIN-SEGMENT>

ZIL library
```

ZILF ignores this and always returns FALSE.

BIND

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

MDL built-in
```

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 GDECL.

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 ...>

ZIL parser library
```

`BIT-SYNONYM` creates synonyms to flag-bits.

Example:

```
<BIT-SYNONYM TAKEBIT GETBIT PICKBIT>
<BIT-SYNONYM LIGHTBIT DAYBIT>
```

BLOAT

```
<BLOAT>

MDL built-in
```

ZILF ignores this and always returns `FALSE`.

`BLOAT` is used in MDL to temporarily expand available storage space to avoid unnecessary garbage collection when loading large files.

BLOCK

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

MDL built-in
```

`BLOCK` pushes current binding of the local `ATOM OBLIST` and rebinds it with the `LIST` of `oblist` supplied as argument and returns the new `<LVAL OBLIST>`.. Usually you want `<ROOT>` as the last `oblist` in `LIST`. `<ENDBLOCK>` then restores the local `ATOM OBLIST` to its previous value.

See *The MDL Programming Language, chapter 15*.

Example:

```
<SETG FOO 111>
<SET BAR 222>
<DEFINE TEST-BLOCK () <PRINT "OUTSIDE BLOCK">>
<BLOCK (<MOBLIST NEW-OBLIST> <ROOT>)>
<SETG FOO 333>
<SET BAR 444>
<DEFINE TEST-BLOCK () <PRINT "INSIDE BLOCK">>
<GVAL FOO>                              -->  333
```

```
<LVAL BAR>                                  -->  444
<TEST-BLOCK>                                -->  "INSIDE BLOCK"
<ENDBLOCK>
<GVAL FOO>                                  -->  111
<LVAL BAR>                                  -->  222
<TEST-BLOCK>                                -->  "OUTSIDE BLOCK"
```

BOUND?

```
<BOUND? atom [environment]>

MDL built-in
```

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 ...>

ZIL parser library
```

BUZZ creates words in the vocabulary with the part-of-speech BUZZ. These are words that can be ignored by the parser or have special handling in the parser.

Example:

```
<BUZZ A AN AND ANY ALL EVERY EVERYTHING BUT EXCEPT OF ONE
      THE THEN UNDO OOPS \. \, \">
```

BYTE

```
<BYTE number>
#BYTE number              ;"Alternative syntax (MDL built-in)"
<CHTYPE number BYTE>      ;"Alternative syntax (MDL built-in)"

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>

ZIL library
```

CHECK-VERSION? is a predicate that returns TRUE if current setting of VERSION is version-spec. Valid values for version-spec are ZIP, EZIP, XZIP, YZIP and the values 3-8.

Examples:

```
<VERSION XZIP>
<CHECK-VERSION? ZIP>      -->  #FALSE
<CHECK-VERSION? 5>        -->  T
```

CHECKPOINT

```
<CHECKPOINT>

ZIL library
```

ZILF ignores this and always returns FALSE.

CHRSET

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

ZIL library
```

CHRSET can be used in version 5+ to replace one or more of the standard character alphabets.

The ZSCII alphabet table is divided up in three blocks of 26 characters each, totaling 78 characters. The default layout is:

```
   Z-char 6789abcdef0123456789abcdef
current   --------------------------
  A0      abcdefghijklmnopqrstuvwxyz
  A1      ABCDEFGHIJKLMNOPQRSTUVWXYZ
  A2       ^0123456789.,!?_#'"/\-:()
          --------------------------
```

Text is then encoded into 2-byte words with 5-bits per character. The left-over bit is always 0 except on the last word where it is 1 to indicate that tis is the last 2-byte word in the text.

```
   --first byte-------   --second byte---
   7    6 5 4 3 2  1 0   7 6 5  4 3 2 1 0
   bit  --first--  --second---  --third--
```

Initially the A0 is the current alphabet. The characters 2, 3, 4 and 5 change alphabet according to

this table:

```
                from A0  from A1  from A2
  Z-char 2        A1       A2       A0
  Z-char 3        A2       A0       A1
  Z-char 4        A1       A2       A0
  Z-char 5        A2       A0       A1
```

Character 2 and 3 change the alphabet for the next character (“shift”) and character 4 and 5 change the alphabet permanent (“shift lock”).

`CHARSET` change one character in one alphabet or change an alphabet altogether.

Example:

```
;"    1         2         3
  67890123456789012345678901
  zyxwvutsrqponmlkjihgfedcba

        z=6   i=23  l=20
      1 00110 10111 10100"

<VERSION 5>

<CHRSET 0 "zyxwvutsrqponmlkjihgfedcba">
<CONSTANT ENCODED-TEXT <TABLE #2 1001101011110100>>
<CONSTANT MYTEXT "zil">

<ROUTINE GO () <TEST-CHRSET>>

<ROUTINE TEST-CHRSET ()
     <PRINTB ,ENCODED-TEXT> <CRLF>
     <PRINT ,MYTEXT> <CRLF>
     <PRINTN <GET ,ENCODED-TEXT 0>> <CRLF>
     <PRINTN <GET <* 4 ,MYTEXT> 0>> <CRLF> ;"Multiply by 4 to
                                            get packed
                                            address in v 5."
     <PRINTN <- <GET <* 4 ,MYTEXT> 0> <GET ,ENCODED-TEXT 0>>>
     <CRLF>>

-->
     zil
     zil
     -25868
     -25868
     0
```

CHTYPE

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

MDL built-in
```

`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>

MDL built-in
```

CLOSE the channel opened by OPEN and returns the channel.

See READSTRING for example.

COMPILATION-FLAG

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

ZIL library
```

This defines a COMPILATION-FLAG named atom-or-string with initialized to value. If no value is supplied it defaults to TRUE. The name of the flag can either be an ATOM or a STRING whose text becomes the ATOM.

The flag can then be read by COMPILATION-FLAG-VALUE or used as a condition in IFFLAG.

A call to COMPILATION-FLAG with an already defined ATOM changes the value of the ATOM.

Examples:

```
<COMPILATION-FLAG MYFLAG>
<COMPILATION-FLAG-VALUE MYFLAG>    -->  T
<COMPILATION-FLAG "MYFLAG" 123>
<COMPILATION-FLAG-VALUE MYFLAG>    -->  123
```

COMPILATION-FLAG-DEFAULT

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

ZIL library
```

This defines a COMPILATION-FLAG named atom-or-string with initialized to value. If no value is supplied it defaults to TRUE. The name of the flag can either be an ATOM or a STRING whose text becomes the ATOM.

The flag can then be read by COMPILATION-FLAG-VALUE or used as a condition in IFFLAG.

A call to COMPILATION-FLAG-DEFAULT with an already defined ATOM doesn't change the value of the ATOM.

Examples:

```
<COMPILATION-FLAG-DEFAULT MYFLAG T>
<COMPILATION-FLAG-VALUE MYFLAG>    -->  T
<COMPILATION-FLAG "MYFLAG" 123>
<COMPILATION-FLAG-VALUE MYFLAG>    -->  123
<COMPILATION-FLAG-DEFAULT MYFLAG T>
<COMPILATION-FLAG-VALUE MYFLAG>    -->  123
```

COMPILATION-FLAG-VALUE

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

ZIL library
```

This returns the value of the COMPILATION-FLAG atom-or-string. If no atom-or-string is defined it returns FALSE.

Examples:

```
<COMPILATION-FLAG MYFLAG 123>
<COMPILATION-FLAG-VALUE MYFLAG>         -->  123
<COMPILATION-FLAG-VALUE ASDFGHJKL>      -->  #FALSE
```

COMPILING?

```
<COMPILING?>

ZIL library
```

ZILF ignores this and always returns TRUE.

Presumably COMPILING? is used in the MDL environment to determine if the game is compiled with ZILCH or running in the interpreter.

COND

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

MDL built-in
```

COND ("conditional") evaluates condition in each (condition body ...) and if the condition is not FALSE it continues to evaluate all the body-parts in this LIST. COND only evaluates the first non-FALSE condition (it ignores the rest) and returns the value of the last performed evaluation.

Examples:

```
;"IF-THEN..."
<COND (<AND <=? 1 1> <=? 2 2>> <PRINC "IF-THEN ..."> <CRLF>)>
;"IF-THEN-ELSE..."
```

```
<COND (<AND <=? 1 1> <=? 2 2>>
     <PRINC "THEN ...">
     <CRLF>
)
(ELSE                                ;"ELSE = T, Catch-all"
     <TELL "ELSE ...">
     <CRLF>
)>

;"IF-THEN-ELSEIF-ELSEIF-ELSE... or SWITCH"
<SET SWITCH 2>
<COND
     (<=? .SWITCH 1>
          <PRINC "Variable SWITCH = 1"> <CRLF>)
     (<=? .SWITCH 2>
          <PRINC "Variable SWITCH = 2"> <CRLF>)
     (<=? .SWITCH 3>
          <PRINC "Variable SWITCH = 3"> <CRLF>)
     (T
          <PRINC "Variable SWITCH not in (1 2 3)"> <CRLF>)
>

;"Trigger on FIRST non-FALSE"
<COND (<SET A <>> <PRINC "Won't execute (always FALSE)">)
      (<SET A 3> <PRINC "Execute (SET returns non-FALSE)">)>
```

CONS

```
<CONS first list>

MDL built-in
```

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 is 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 built-in
```

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>

MDL built-in
```

DECL-CHECK turns off or on type declaration checking. It is initially on.

Examples:

```
<DECL-CHECK <>>
<GDECL (FOO) FIX>
<SETG FOO <>>                  -->  Ok!

<DECL-CHECK T>
<SETG FOO <>>                  -->  Error
```

DECL?

```
<DECL? value pattern>

MDL built-in
```

Predicate. DECL? returns TRUE if value checks against pattern, otherwise FALSE. For the format of the pattern, see GDECL.

Examples:

```
;"Simple DECL"
<DECL? 1 FIX>                                    -->  T
<DECL? "hi" STRING>                              -->  T
<DECL? FOO STRING>                               -->  #FALSE

;"OR DECL"
```

```
<DECL? 1 '<OR FIX FALSE>>                                  -->  T
<DECL? "hi" '<OR VECTOR STRING>>                           -->  T
<DECL? FOO '<OR STRING FIX>>                               -->  #FALSE

;"Structure DECL"
<DECL? '(1) '<LIST FIX>                                    -->  T
<DECL? '(1) '<LIST ATOM>>                                  -->  #FALSE
<DECL? '<1> '<LIST FIX>>                                   -->  #FALSE
<DECL? '<1> '<<OR FORM LIST> FIX>>                         -->  T
<DECL? '<1> '<<OR <PRIMTYPE LIST> <PRIMTYPE STRING>> FIX>>
                                                           -->  T
<DECL? '(1) '<<PRIMTYPE LIST> FIX>>                        -->  T
<DECL? '<1> '<<PRIMTYPE LIST> FIX>>                        -->  T

;"NTH DECL"
<DECL? '["hi" 456 789 1011] '<VECTOR STRING [4 FIX]>>
                                                           -->  #FALSE
<DECL? '["hi" 456 789 1011] '<VECTOR STRING [3 FIX]>>
                                                           -->  T
<DECL? '["hi" 456 789 1011] '<VECTOR [3 FIX]>>             -->  #FALSE
<DECL? '["hi" 456 789 1011] '<VECTOR STRING [2 FIX]>>
                                                           -->  T
<DECL? '["hi" 456 789 1011] '<VECTOR STRING [2 FIX] FIX>>
                                                           -->  T
<DECL? '["hi" 456 789 1011] '<VECTOR STRING [2 FIX] ATOM>>
                                                           -->  #FALSE
<DECL? '(1 MONEY 2 SHOW 3 READY 4 GO) '<LIST [4 FIX ATOM]>>
                                                           -->  T
<DECL? '(1 MONEY 2 SHOW 3 READY 4 GO) '<LIST [4 FIX]>>
                                                           -->  #FALSE
<DECL? '(1 MONEY 2 SHOW 3 READY 4 GO)
               '<LIST [3 FIX ATOM] FIX ATOM>>              -->  T
<DECL? '(1 MONEY 2 SHOW 3 READY 4 GO) '<LIST [3 FIX ATOM]>>
                                                           -->  T

;"REST DECL"
<DECL? '["hi" 456 789 1011] '<VECTOR STRING FIX [REST FIX]>>
                                                           -->  T
<DECL? '(FOO BAR) '<LIST STRING [REST FIX]>>               -->  #FALSE
<DECL? '(FOO BAR) '<LIST ATOM [REST FIX]>>                 -->  #FALSE
<DECL? '(FOO BAR) '<LIST ATOM ATOM [REST FIX]>>            -->  T

;"OPT DECL"
<DECL? '(FOO BAR) '<LIST [OPT FIX FIX] [REST ATOM]>>
                                                           -->  T
<DECL? '(1 FOO BAR) '<LIST [OPT FIX FIX] [REST ATOM]>>
                                                           -->  T
<DECL? '(1 2 FOO BAR) '<LIST [OPT FIX] [REST ATOM]>>
                                                           -->  #FALSE
```

```
<DECL? '(1 2 FOO BAR) '<LIST [OPT FIX FIX] [REST ATOM]>>
                                                    -->  T
<DECL? '(1 2) '<LIST [OPT FIX FIX] [REST ATOM]>>  -->  T

;"QUOTE DECL"
<DECL? FOO ''FOO>                                   -->  T
<DECL? FOO ''BAR>                                   -->  #FALSE
<DECL? '<OR FIX FALSE> ''<OR FIX FALSE>>            -->  T
<DECL? 123 ''<OR FIX FALSE>>                        -->  #FALSE

;"Segment DECL"
<DECL? '(1 2 3) '<LIST FIX FIX>>                    -->  T
<DECL? '(1 2 3) '!<LIST FIX FIX>>                   -->  #FALSE
<DECL? '(1 2) '!<LIST FIX FIX>>                     -->  T
<DECL? '(1 2) '!<LIST [REST FIX FIX]>>              -->  T
<DECL? '(1 2 3) '!<LIST [REST FIX FIX]>>            -->  #FALSE
<DECL? '(1 2 3 4) '!<LIST [REST FIX FIX]>>          -->  T

;"LVAL/GVAL DECL"
<DECL? '.X LVAL>                                    -->  T
<DECL? '.X GVAL>                                    -->  #FALSE
<DECL? ',X GVAL>                                    -->  T
<DECL? ',X LVAL>                                    -->  #FALSE
<DECL? '.X '<PRIMTYPE ATOM>>                        -->  T
<DECL? ',X '<PRIMTYPE ATOM>>                        -->  T
```

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).

See `DELAY-DEFINITION` and `REPLACE-DEFINITION`.

Examples:

```
<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"
```

DEFAULTS-DEFINED

```
<DEFAULTS-DEFINED>

ZIL library
```

ZILF ignores this and always returns `FALSE`.

DEFINE

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

MDL built-in
```

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 a 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
<POWER-TO 3 0>                          -->  1
```

DEFINE-GLOBALS

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

ZIL libary
```

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.)

DEFINE-SEGMENT

```
<DEFINE-SEGMENT>

ZIL library
```

ZILF ignores this and always returns `FALSE`.

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>

MDL package system
```

DEFINITIONS is exactly as PACKAGE except that there is no internal OBLIST with DEFINITIONS, every ATOM created inside the DEFINITIONS is on the external OBLIST automatically.

To activate a package-name INCLUDE or INCLUDE-WHEN is used.

See END-DEFINITIONS, INCLUDE, INCLUDE-WHEN, PACKAGE and RENTRY.

Examples:

```
;"Define PACKAGE"
<REMOVE ANSWER> ;"Secure that ATOM not on any OBLIST"
<DEFINITIONS "FOO">
<SETG ANSWER 42>
<END-DEFINITIONS>

<TYPE? <GETPROP FOO!-PACKAGE OBLIST> OBLIST>          -->  OBLIST
<GASSIGNED? ANSWER>                                  -->  #FALSE
<GASSIGNED? ANSWER!-FOO!-PACKAGE>                    -->  T
,ANSWER!-FOO!-PACKAGE                                -->  42

<REMOVE ANSWER> ;"Secure that ATOM not on any OBLIST"
<INCLUDE "FOO">
,ANSWER                                              -->  42
```

DEFMAC

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

MDL built-in
```

DEFMAC has the same syntax as DEFINE, but defines a MACRO instead of a FUNCTION. A MACRO is evaluated two times, the first evaluation inserts the arguments in the MACRO and creates an object that is evaluated during the second evaluation. The first evaluation is done at “top-level”, in other words during compilation. EXPAND is used to perform the first evaluation.

Note that two identical calls to a MACRO always generate the same result from the first evaluation.

Example:

```
<DEFMAC INC (ATM "OPTIONAL" (N 1))
     <FORM SET .ATM
          <FORM + <FORM LVAL .ATM> .N>>>
<SET X 1>
<INC X 2>                                  -->  3
<EXPAND '<INC X 2>>                        -->  <SET X <+ .X 2>>
```

DEFSTRUCT

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

MDL built-in
```

DEFSTRUCT creates abstract record structures and creates an user-defined TYPE. In practice DEFSTRUCT builds a couple of MACROs that can be used to create and access instances of the structure.

type-name is the name of the new structure-type. There will also be a new TYPE with type-name and the MACRO, MAKE-[type-name] is created that can be used to create new instances of this new TYPE.

The base-type is the container TYPE for this new structure-type. It is usually a VECTOR, LIST or TABLE. The struct-options are used to change the default behaviour of the base-type. It is possible to change the default behaviour with SET-DEFSTRUCT-DEFAULT-FILE. The struct-options are:

'CONSTRUCTOR	Defines a new constructor for the MAKE- MACRO. See examples below on how to define a new constructor. If there is no definition specified after 'CONSTRUCTOR no MACRO will be created for this type-name.
'INIT-ARGS	Defines if there are arguments when the base-type is created. For example (PURE) when using TABLE as a container. Default is that there are no arguments.
'NODECL	Specifies that no TYPE-checking should occur when storing values in container base-type. Default is that there is TYPE-checking.
'NOTYPE	Specifies that no new TYPE is created for this type-name. Default is that a new TYPE is created.

Then follows all the fields in the record. Every field has a field-name and a TYPE declaration, decl. The decl follows the ordinary syntax for declarations, see GDECL. The field-options are optional, but they can be one or more of these:

default-value	A default value for this field.
'NTH	FUNCTION to read fields from the container. Default FUNCTION is NTH.
'OFFSET	Index in container to store this value. FUNCTION is NTH.
'PUT	FUNCTION to insert value in this field in the container. Default FUNCTION is PUT.

For every field in the record there is a MACRO created with the field-name that can be used to read and insert values in the field.

Examples:

```
;"Create new object"
<DEFSTRUCT BOOK LIST (TITLE STRING) (AUTHOR STRING)
                     (SUBJECT STRING) (BOOK-ID FIX)>
```

```
<SET BOOK1 <MAKE-BOOK 'TITLE "C Programming"
                      'AUTHOR "Nuha Ali"
                      'SUBJECT "C-Programming Tutorial"
                      'BOOK-ID 478>>
<SET BOOK2 <MAKE-BOOK 'TITLE "Telecom Billing"
                      'AUTHOR "Zara Ali"
                      'SUBJECT "C-Programming Tutorial"
                      'BOOK-ID 501>>
<TITLE .BOOK1>                 -->  "C Programming"
<TYPE .BOOK1>                  -->  BOOK
<1 .BOOK2>                     -->  "Telecom Billing"
<AUTHOR .BOOK2 "Paul Auster">
<AUTHOR .BOOK2>                -->  "Paul Auster"
<MAKE-BOOK>                    -->  ("" "" "" 0)

;"Put values into existing object"
<DEFSTRUCT POINT VECTOR (POINT-X FIX) (POINT-Y FIX)>
<SET MY-VECTOR [123 456 789 1011]>
<MAKE-POINT 'POINT .MY-VECTOR 'POINT-Y 999 'POINT-X 888>
                               -->  [888 999 789 1011]

;"Use field value-default and offset"
<DEFSTRUCT RPOINT VECTOR (RPOINT-X FIX 'OFFSET 2)
                         (RPOINT-Y FIX 456 'OFFSET 1)>
<MAKE-RPOINT 'RPOINT-X 123>    -->  #RPOINT [456 123]
<RPOINT-Y #RPOINT [234 567]>   -->  234

;"Create struct without creating TYPE"
<DEFSTRUCT PTNT (VECTOR 'NOTYPE) (PTNT-X FIX) (PTNT-Y FIX)>
<VALID-TYPE? PTNT>             -->  #FALSE
<PTNT-X [123 456]>             -->  123

;"Create struct without declaration checks"
<DEFSTRUCT PTND (VECTOR 'NODECL) (PTND-X FIX) (PTND-Y FIX)>
<CHTYPE [FOO BAR] PTND>        -->  [FOO BAR]
<CHTYPE [FOO BAR] POINT>       -->  ERROR

;"Create struct with suppressed constructor"
<DEFSTRUCT P-C (VECTOR 'CONSTRUCTOR) (P-C FIX) (P-C-Y FIX)>
<GASSIGNED? MAKE-P-C>          -->  #FALSE
<VALID-TYPE? P-C>              -->  P-C

;"Create struct with INIT-ARGS"
<DEFSTRUCT PT-TBL (TABLE ('INIT-ARGS (PURE)))
     (PT-TBL-X FIX) (PT-TBL-Y FIX)>
<MAKE-PT-TBL 123 456>    --> #PT-TBL %<TABLE (PURE) 123 456>

;"Positional constructor arguments"
<DEFSTRUCT FOO VECTOR (FOO-A ATOM) (FOO-B <OR FIX FALSE>)>
<MAKE-FOO BAR>                 -->  #FOO [BAR #FALSE ()]

;"Custom constructor"
<SETG NEXT-ID 0>
```

```
<DEFSTRUCT RGBA (VECTOR 'CONSTRUCTOR
     ('CONSTRUCTOR MAKE-RGBA
     ('RED 'GREEN 'BLUE "OPT" ('ALPHA 255)
     "AUX" (RGBA-ID '<SETG NEXT-ID <+ ,NEXT-ID 1>>))))
     (RED FIX) (GREEN FIX) (BLUE FIX) (ALPHA FIX)
     (RGBA-ID FIX)>
<RED <MAKE-RGBA 10 20 30>>          -->  10
<RGBA-ID <MAKE-RGBA 11 22 33>>      -->  1
<ALPHA <MAKE-RGBA 11 22 33>>        -->  255
<ALPHA <MAKE-RGBA 11 22 33 44>>     -->  44

;"Eval or not to eval arguments"
<DEFSTRUCT E-PT VECTOR (E-PT-X FIX) (E-PT-Y <OR FIX FORM>)>
<MAKE-E-PT <+ 1 2> '<+ 3 4>>        -->  #E-PT [3 <+ 3 4>]

;"Explicit default values"
<DEFSTRUCT PT2 VECTOR (PT2-X FIX 123) (PT2-Y FIX 456)
                      (PT2-ID FIX <ALLOCATE-ID>)>
<SETG NEXT-ID 1>
<DEFINE ALLOCATE-ID ("AUX" (R ,NEXT-ID))
     <SETG NEXT-ID <+ ,NEXT-ID 1>> .R>
<PT2-ID <MAKE-PT2>>                 -->  1
<PT2-ID <MAKE-PT2>>                 -->  2
<PT2-ID <MAKE-PT2 'PT2-ID 1001>>    -->  1001
<PT2-ID <MAKE-PT2>>                 -->  3
<PT2-X <MAKE-PT2 'PT2-Y 0>>         -->  123
<PT2-Y <MAKE-PT2 'PT2-Y 0>>         -->  0
<PT2-ID <MAKE-PT2>>                 -->  6
```

DELAY-DEFINITION

```
<DELAY-DEFINITION name>

ZIL library
```

DELAY-DEFINITION tells ZILF that a REPLACE-DEFINITION for name should be expected thus the DEFAULT-DEFINITION never is evaluated for the name. This means that REPLACE-DEFINITION can appear after the DEFAULT-DEFINITION.

DELAY-DEFINITION also means that the body of REPLACE-DEFINITION will be evaluated at the place of REPLACE-DEFINITION.

See DEFAULT-DEFINITION and REPLACE-DEFINITION.

Examples:

```
;"REPLACE can be defined after DEFAULT"
<DELAY-DEFINITION FOO-ROUTINE>
<DEFAULT-DEFINITION FOO-ROUTINE <DEFINE FOO () 123>>
<REPLACE-DEFINITION FOO-ROUTINE <DEFINE FOO () 456>>

<FOO>                       -->  456
```

```
;"DELAY means that REPLACE is evaluated at right place"
<DELAY-DEFINITION BAR-ROUTINE>
<SETG BAR-RESULT 789>
<REPLACE-DEFINITION BAR-ROUTINE
     <EVAL <FORM DEFINE BAR '() ,BAR-RESULT>>>
<SETG BAR-RESULT 123>
<DEFAULT-DEFINITION BAR-ROUTINE
     <EVAL <FORM DEFINE BAR '() ,BAR-RESULT>>>
<BAR>                    -->  789 ;"123 without DELAY"
```

DIR-SYNONYM

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

ZIL parser library
```

`DIR-SYNONYM` creates one or more `synonyms` to the `original` direction.

ZILF treats `DIR-SYNONYM` as an alias to `SYNONYM`.

DIRECTIONS

```
<DIRECTIONS atoms ...>

ZIL parser library
```

`DIRECTIONS` creates words in the vocabulary with the `part-of-speech` `DIRECTION`. `DIRECTIONS` are often defined in the parser and the order is usually tightly tied to the parser. Be careful if you change these. You can use `DIR-SYNONYM` if you, for example, want to add `FORE`, `AFT`, `PORT` and `STARBOARD`.

Example:

```
<DIRECTIONS NORTH SOUTH EAST WEST NE NW SE SW IN OUT UP DOWN>
```

EMPTY?

```
<EMPTY? structure>

MDL built-in
```

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>
```

```
MDL package system
```

END-DEFINITIONS is an alias to ENDBLOCK.

See DEFINITIONS.

END-SEGMENT

```
<END-SEGMENT>

ZIL library
```

ZILF ignores this and always returns FALSE.

ENDBLOCK

```
<ENDBLOCK>

MDL built-in
```

ENDBLOCK pops back, rebinds and returns the local ATOM OBLIST to the state it had before the call to BLOCK. ENDBLOCK without previous BLOCK (or PACKAGE, DEFINITIONS, etc) results in an error.

See *The MDL Programming Language, chapter 15*.

Example:

```
XYZZY!-MY-OBLIST
<SETG FIRST!- FOO>
<BLOCK (<GETPROP MY-OBLIST OBLIST> <ROOT>)>
<SETG SECOND!- FOO>
<ENDBLOCK>
<=? ,FIRST!- ,SECOND!->                    -->  #FALSE
```

ENDLOAD

```
<ENDLOAD>

ZIL library
```

ZILF ignores this and always returns FALSE.

ENDPACKAGE

```
<ENDPACKAGE>

MDL package system
```

ENDPACKAGE is an alias to ENDBLOCK.

See PACKAGE.

ENDSECTION

```
<ENDSECTION>

MDL package system
```

ENDSECTION is an alias to ENDBLOCK.

See DEFINITIONS.

ENTRY

```
<ENTRY atoms ...>

MDL package system
```

ENTRY creates/moves one or more ATOMs to the external OBLIST in a PACKAGE. ENTRY is only valid inside a PACKAGE, if it's used outside an error is raised.

See PACKAGE, RENTRY, USE, USE-WHEN.

Examples:

```
<REMOVE ANSWER> ;"Secure that ATOM not on any OBLIST"
<PACKAGE "FOO">
<SETG ANSWER 42>
<1 .OBLIST>    -->  #OBLIST (("ANSWER" ANSWER))
<2 .OBLIST>    -->  #OBLIST (("IFOO" IFOO))
<ENTRY ANSWER>
<1 .OBLIST>    -->  #OBLIST ()
<2 .OBLIST>    -->  #OBLIST (("IFOO" IFOO) ("ANSWER" ANSWER))
<ENDPACKAGE>
,ANSWER        -->  42
```

EQVB

```
<EQVB numbers ...>

MDL built-in
```

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 ...>
```

```
MDL built-in
```

ERROR raises an error ([error MDL0001]) and listing values as resources. The values are usually a text explaining the error, offending ATOM, routine where it occurred and last any other information.

Example:

```
<SET A 616>
<ERROR "MY TYPE OF ERROR." .A>

-->
[error MDL0001] <stdin>:1: ERROR: "MY TYPE OF ERROR." 616
```

EVAL

```
<EVAL value [environment]>

MDL built-in
```

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
```

EVAL-IN-SEGMENT

```
<EVAL-IN-SEGMENT dummy1 value[dummy2]>

ZIL library
```

ZILF ignores dummy1 and the optional dummy2. ZILF then calls EVAL on the value and returns its result.

Example:

```
<SET F '<+ 1 2>>

.F                                          -->    <+ 1 2>
<EVAL-IN-SEGMENT "HINTS" .F (1 2 3)>        -->  3
```

EVALTYPE

```
<EVALTYPE atom [handler]>

MDL built-in
```

EVALTYPE tells the TYPE atom how it should be evaluated by EVAL. If EVALTYPE is called without a handler then the currently active handler is returned. If there is no active handler, FALSE is returned.

Note that it is possible to replace the handler with a new handler, even on the predefined TYPEs.

See APPLYTYPE, NEWTYPE and PRINTTYPE.

Example:

```
<NEWTYPE GRITCH LIST>
<EVALTYPE GRITCH>                          -->  #FALSE
<EVALTYPE GRITCH LIST> ;"Evaluate GRITCH as a LIST"
<EVALTYPE GRITCH>                          -->  LIST
#GRITCH (A <+ 1 2 3> !<SET A "BC">)        -->  (A 6 !\B !\C)

;"Make it like LISP!"
<EVALTYPE LIST FORM>  ;"Evaluate LISTs as FORMs!"
<EVALTYPE ATOM ,LVAL> ;"Evaluate bare ATOM as LVAL!"
(+ 1 2)                                    -->  3
(SET 'A 5)
A                                          -->  5
```

EXPAND

```
<EXPAND value>

MDL built-in
```

EXPAND performs the first EVAL of the value. In case the value is a MACRO only the first EVAL is done.

Example:

```
<DEFMAC INC2 (ATM "OPTIONAL" (N 1))
     <PARSE "<SET %.ATM <+ %.ATM %.N>>">>
<EXPAND '<INC2 X>>             -->  <SET X <+ X 1>>
```

FILE-FLAGS

```
<FILE-FLAGS {CLEAN-STACK? | MDL-ZIL? | SENTENCE-ENDS? |
                ZAP-TO-SOURCE-DIRECTORY?} ...>

ZIL library
```

This sets flags to control how ZILF should compile. To clear, call FILE-FLAGS without any flags.

The flags are:

`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.
`MDL-ZIL?`	Tells the compiler to treat `SETG` (at top-level) as `GLOBAL` and `DEFINE` as `ROUTINE` (`SETG20` and `DEFINE20` always works as in MDL). Presumably that was a way to run the games in MDL during development without recompiling them.
`SENTENCE-ENDS?`	Tells the compiler (only version 6) to treat two spaces after a period or a question mark as the end of a sentence in TELL. Note: a space followed by an embedded newline wil produce two spaces instead of collapsing.
`ZAP-TO-SOURCE-DIRECTORY?`	ZILF ignores this.

Examples:

```
<FILE-FLAGS CLEAN-STACK? MDL-ZIL?> --> Set both flags

<FILE-FLAGS MDL-ZIL?>
<SETG X 123>                          ;"This compiles as GLOBAL"
<DEFINE MDL-ZIL-TEST () <TELL N X CR>>  ;"This compiles as a
                                          ROUTINE"

<FILE-FLAGS SENTENCE-ENDS?>
<ROUTINE SENTENCE-ENDS-TEST ()
     <TELL \"Hi.  Hi.   Hi.|  Hi!  Hi?  Hi. \nHi.\" CR>">
-->  "Hi.\u000bHi.\u000b Hi.\n  Hi!\u000bHi?\u000bHi.  Hi.\n"
```

FILE-LENGTH

```
<FILE-LENGTH channel>

MDL built-in
```

`FILE-LENGTH` returns the size, in bytes, of the file on `channel`. `FILE-LENGTH` returns `FALSE` if the file is closed.

Example:

```
;"ZILF ver 0.9"
<SET CH <OPEN "READ" "../zillib/parser.zil">>
<FILE-LENGTH .CH>                        -->  115629
<CLOSE .CH>
```

FLOAD

```
<FLOAD filename>

MDL built-in
```

ZILF ignores all but the first argument and treats FLOAD as an alias to INSERT-FILE.

FORM

```
<FORM values ...>

MDL built-in
```

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>
```

FREQUENT-WORDS?

```
<FREQUENT-WORDS?>

ZIL library
```

ZILF ignores this and always returns FALSE. Frequent words table is built by ZAPF instead.

FUNCTION

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

MDL built-in
```

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 ATOM that binds the ATOM to the ENVIRONMENT when the FUNCTION was applied. See EVAL for example on this.
Arguments	The required arguments for this FUNCTION. The arguments are bound to local variables inside this FUNCTION.
"OPT"	The optional arguments for this FUNCTION. The arguments are bound to local variables inside this FUNCTION and can be defined with a default value. "OPTIONAL" is an alias for "OPT".
"ARGS"	Followed by an ATOM that is bound a LIST of all remaining arguments, unevaluated. If "ARGS" appears in arg-list, "TUPLE" should not appear.
"TUPLE"	Followed by an ATOM that is bound a TUPLE of all remaining arguments, evaluated. If "TUPLE" appears in arg-list, "ARGS" should not appear. See TUPLE for example on this.

"AUX"	Followed by any number of ATOMs that becomes local variables inside this FUNCTION and can be defined with a default value. "EXTRA" is a alias for "AUX".
"NAME"	Followed by an ATOM that becomes the activation for this FUNCTION. This is equivalent to naming the activation before the arg-list. "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.

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 built-in
```

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

G?

```
<G? value1 value2>

MDL built-in
```

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

GASSIGNED?

```
<GASSIGNED? Atom>

MDL built-in
```

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 built-in
```

`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 built-in
```

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
```

GC-MON

```
<GC-MON>

MDL built-in
```

ZILF ignores this and always returns FALSE.

GDECL

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

MDL built-in
```

GDECL declares the type/structure of the global value of ATOMs. GDECL pairs a LIST of atoms with a decl pattern, this can then be repeated indefinitely.

The decl pattern can contain the following:

A TYPE name	The atoms TYPE must be of this TYPE. This can be generalized slightly by using <PRIMTYPE type>, which means that the atoms TYPE must have the same PRIMTYPE as type.
ANY	The atom can be of any TYPE.
STRUCTURED	Means that <STRUCTURED? atom> must be TRUE (atom is for example a LIST, VECTOR or STRING).
APPLICABLE	Means that <APPLICABLE? atom> must be TRUE (atom is for example a FIX, FUNCTION or MACRO).
A QUOTEd ATOM	Means that the atom must be =? with the QUOTEd ATOM.

If the decl pattern is STRUCTURED it is possible to specify a pattern for the structure. This has the following syntax:

<structure patterns ...>	This means that the structure must follow the defined pattern (so long it is defined). Items in the structure at positions beyond the defined pattern can be of any TYPE.

This means that, for example, <GDECL (X) <LIST FIX ANY FIX>> is declaring that X must be a LIST (at least of LENGTH 3), with a FIX in position 1 and 3 and any TYPE in position 2 and position 4 and beyond.

```
<SETG X (1 2 3)>          is legal
<SETG X (1 2 3 4)>        is legal
<SETG X (1 2 3 !\A)>      is legal
<SETG X (1 2)>            is illegal
```

```
<SETG X (!\A 2 3)>          is illegal
```

Normally the pattern for structures defines that the structure should at least contain these elements, but it can contain additional items. If you want to disallow additional items , a SEGMENT is used instead of a FORM. <GDECL (X) !<LIST FIX ANY FIX>> means that the LIST must have exactly LENGTH 3.

```
<SETG X (1 2 3)>            is legal
<SETG X (1 2 3 4)>          is illegal
<SETG X (1 2 3 !\A)>        is illegal
<SETG X (1 2)>              is illegal
<SETG X (!\A 2 3)>          is illegal
```

The pattern in this construction can in turn be defined to repeat itself by the syntax:

`[number patterns ...]`	Means that specified `pattern` should repeat itself `number` of times.
`[REST patterns ...]`	Means that specified `pattern` should repeat itself indefinitely. If this is defined it must be the last in the structure declaration.
`[OPT patterns ...]`	Means that this `structure` can either be empty or follow the defined `pattern`. Only a `REST` construction can follow `OPT`.

Finally, it is allowed to specify several possible `decl` to an atom with the compound `decl OR`.

`<OR decl ...>`	This means that the `atoms` can be one of the specified `decl`. Each of the `decl` follow the same rules as above.

Examples:

```
                                        X must be:
<GDECL (X) FIX>                 -->  FIX
<GDECL (X) <OR FIX STRING>>     -->  FIX or STRING
<GDECL (X) <LIST FIX>           -->  LIST with FIX in pos 1
<GDECL (X) <LIST [3 FIX]>       -->  LIST with FIX in pos 1-3
<GDECL (X) <LIST [REST FIX]>    -->  LIST with only FIX
<GDECL (X) <LIST [OPT FIX] [REST FIX]>>
                     -->  Empty LIST or LIST containing FIX

See DECL? for more examples on how to format decl.
```

GET-DECL

```
<GET-DECL item>

MDL built-in
```

`GET-DECL` returns the `pattern` defined to the `item`. It returns `FALSE` if no `item` exists.

See `DECL?`, `GDECL` and `PUT-DECL` for more on declaration patterns.

Examples:

```
<GET-DECL BOOLEAN>                      -->  #FALSE
```

```
<PUT-DECL BOOLEAN '<OR ATOM FALSE>>
<GET-DECL BOOLEAN>                        -->  <OR ATOM FALSE>
```

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 built-in `GETB`.

Also see `PUTB`, `ZGET`, `ZPUT` and `ZREST`.

Example:

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

GETPROP

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

MDL built-in
```

`GETPROP` returns the property-value stored under `indicator` on `item`. If no value can be found `GETPROP` returns the default-value or `FALSE` if no default-value is given.

See `ASSOCIATIONS`, `AVALUE`, `INDICATOR`, `ITEM`, `NEXT` and `PUTPROP`.

Examples:

```
<PUTPROP FOO BAR BAZ>
<GETPROP FOO BAR>                     --> BAZ
<GETPROP FOO BAZ>                     --> #FALSE
<GETPROP FOO BAZ "Value not found."> --> "Value not found."

<SET L (1 2 3)>
<PUTPROP .L FOO 456>
.L                                    --> (1 2 3)
<GETPROP .L FOO>                      --> 456
```

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 vector end beginning>

MDL built-in
```

GROW expands the vector with end and/or beginning number of elements to respectively end of the vector. Only non-negative values for end and beginning are valid. The new elements have FALSE as an initial value.

If elements are added to the beginning of a vector all old references to that vector have to use TOP or BACK to access the new elements.

Examples:

```
<SET V1 [1 2 3]>
<SET V2 <GROW .V1 1 1>>
<LVAL V1>                 -->  [1 2 3 #FALSE ()]
<LVAL V2>                 -->  [#FALSE () 1 2 3 #FALSE ()]
<2 .V1 4>
<LVAL V1>                 -->  [1 4 3 #FALSE ()]
<LVAL V2>                 -->  [#FALSE () 1 4 3 #FALSE ()]
<TOP .V1>                 -->  [#FALSE () 1 4 3 #FALSE ()]
```

GUNASSIGN

```
<GUNASSIGN atom>

MDL built-in
```

Unassign global atom.

Example:

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

GVAL

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

MDL built-in
```

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 `ATOM`s whose `PNAME`s 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`.

Example:

```
<COMPILATION-FLAG MYFLAG <>>
<IFFLAG (MYFLAG <SETG FOO "NOT OFF">) (T <SETG FOO "OFF">)>
,FOO           -->  "OFF"
```

ILIST

```
<ILIST count [init]>

MDL built-in
```

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]>

MDL built-in
```

`IMAGE` prints the actual raw character with number `ch` to `channel`. No extra characters are ever

printed. IMAGE returns ch.

Example:

```
<DEFINE FOO ()
     <IMAGE 70>
     <IMAGE 79>
     <IMAGE 79>
     <CRLF>>

<FOO>                --> "FOO"
```

INCLUDE

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

MDL package system
```

INCLUDE activates one or many package-names and makes its content available in the current OBLIST-path. In practice INCLUDE copies the OBLIST package-name and adds it last to the local OBLIST (<LVAL OBLIST>). This means that all ATOMs on the DEFINITIONS OBLIST becomes available in current environment.

If the package-name is not available in the current environment, INCLUDE tries to load "package-name.zil" from the current path.

INCLUDE only works together with DEFINITIONS and if the definition of the package-name is missing from the environment or no file is found containing that definition is found, an error is raised.

See DEFINITIONS and INCLUDE-WHEN.

Example:

```
<INCLUDE "FOOFOO">  ;"Searches for file "foofoo.zil" which
                    contains the definition for
                    <DEFINITIONS "FOOFOO"> ..."
```

INCLUDE-WHEN

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

MDL package system
```

INCLUDE-WHEN is exactly like INCLUDE but only activates the package-name if the condition evaluates to TRUE.

See DEFINITIONS and INCLUDE.

Example:

```
<DEFINITIONS "FOO">
<SETG AAAA 1234>
<END-DEFINITIONS>

<GASSIGNED? AAAA>                                  -->  #FALSE
<REMOVE AAAA> ;"Secure that ATOM not on any OBLIST"
```

```
<INCLUDE-WHEN <=? 1 2> "FOO">
<GASSIGNED? AAAA>                                    -->  #FALSE
<REMOVE AAAA> ;"Secure that ATOM not on any OBLIST"
<INCLUDE-WHEN <=? 1 1> "FOO">
,AAAA                                                -->  1234
```

INDENT-TO

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

ZIL library
```

`INDENT-TO` places the cursor at the `position` on `channel`. Default value for the `channel` is `.OUTCHAN` (the console).

Example:

```
<DEFINE PRINT-2-COL (LST)
     <REPEAT ((I 0))
          <SET I <+ .I 1>>
          <COND (<G? .I <LENGTH .LST>> <RETURN>)>
          <COND (<1? <MOD .I 2>>
                     <INDENT-TO 3>
                     <PRINC <.I .LST>>)
                (T   <INDENT-TO 15>
                     <PRINC <.I .LST>>
                     <CRLF>)>>
     <CRLF>>
<PRINT-2-COL ("Apple" "Banana" "Orange" "Lime")>
     -->     Apple       Banana
             Orange      Lime
```

INDEX

```
<INDEX offset>

MDL built-in
```

`INDEX` returns the `index`-part of an `OFFSET`.

Example:

```
<SETG OFF3 <OFFSET 3 '<VECTOR> 'STRING>>
<INDEX ,OFF3>                                  -->  3
```

INDICATOR

```
<INDICATOR asoc>

MDL built-in
```

`INDICATOR` returns the indicator part from an `asoc` entry, of `TYPE ASOC`, in the `ASSOCIATION`

chain.

See ASSOCIATIONS, AVALUE, GETPROP, ITEM, NEXT and PUTPROP.

Example:

```
<DEFINE LAST-ASOC ()
     <REPEAT ((A <ASSOCIATIONS>))
          <COND (<=? .A <>> <RETURN <>>)
          (<=? <NEXT .A> <>> <RETURN .A>)>
     <SET A <NEXT .A>>>>

<PUTPROP NEW-ASOC TEXT "Hello, world!">
<SET A <LAST-ASOC>>
<INDICATOR .A>                --> TEXT
```

INSERT

```
<INSERT atom | pname oblist>

MDL built-in
```

INSERT creates an ATOM with the pname and inserts it into oblist. INSERT returns the newly created ATOM (or raises an error if the ATOM already was on the oblist). First argument can also be an atom but this ATOM can not be on any OBLIST and therefore must be newly created by ATOM or recently REMOVEd.

INSERT requires that you specify oblist, if you want to create an ATOM on the standard OBLIST, usually <1 .OBLIST>, you can use <PARSE string> instead.

Note that you also can use trailers to both create the ATOM and the OBLIST (or one of them). atom!-oblist inserts the atom on the oblist and if one of them or both don't exist, they are created.

See *The MDL Programming Language, chapter 15*.

Examples:

```
<INSERT "FOO-1" <MOBLIST OB>>                 -->  FOO-1!-OB

<INSERT <ATOM "FOO-2"> <MOBLIST OB>>          -->  FOO-2!-OB
<INSERT <REMOVE "FOO-2" <MOBLIST OB>> <MOBLIST OB2>>
                                              -->  FOO-2!-OB2

<INSERT FOO-3 <MOBLIST OB>>
     -->  Error (Interpreter already placed it on <1 .OBLIST>

;"Returns FOO from OB. Creates ATOM/OBLIST if needed."
<OR <LOOKUP "FOO" <MOBLIST OB>> <INSERT "FOO" <MOBLIST OB>>
                                              -->  FOO!-OB

FOO!-OB                                       -->  FOO!-OB
BAR!-OB                                       -->  BAR!-OB
<MOBLIST OB>   -->  #OBLIST (("FOO" FOO!-OB) ("BAR" BAR!-OB))
```

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.

ZILF ignores all but the first argument.

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 built-in
```

`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>

MDL built-in
```

ITEM returns the item part from an asoc entry, of TYPE ASOC, in the ASSOCIATION chain.

See ASSOCIATIONS, AVALUE, GETPROP, INDICATOR, NEXT and PUTPROP.

Example:

```
<DEFINE LAST-ASOC ()
     <REPEAT ((A <ASSOCIATIONS>))
          <COND (<=? .A <>> <RETURN <>>)
          (<=? <NEXT .A> <>> <RETURN .A>)>
     <SET A <NEXT .A>>>>

<PUTPROP NEW-ASOC TEXT "Hello, world!">
<SET A <LAST-ASOC>>
<ITEM .A>                 -->  NEW-ASOC
```

IVECTOR

```
<IVECTOR count [init]>

MDL built-in
```

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 built-in
```

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

L?

```
<L? value1 value2>

MDL built-in
```

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

LEGAL?

```
<LEGAL? value>

MDL built-in
```

`LEGAL?` is a predicate that returns `TRUE` if portion of the stack value occupies is still active, otherwise `FALSE`. Although `LEGAL?` works for all `TYPE`s, it's only useful for those `TYPE`s that live on the stack, like `TUPLE`, `activation` and `environment`, all other types always return

TRUE.

Examples:

```
;"Activation"
<DEFINE FOO ACT () <SETG ACT .ACT> <LEGAL? .ACT>>
<FOO>           -->  T      ;"ACT legal inside function"
<LEGAL? ,ACT>   -->  #FALSE ;"ACT illegal outside function"

;"Environment"
<DEFINE BAR () <BAZ>>
<DEFINE BAZ ("BIND" ENV) <SETG ENV .ENV> <LEGAL? .ENV>>
<BAR>           -->  T      ;"Sets ENV to BARs environment"
<LEGAL? ,ENV>   -->  #FALSE ;"BARs environment illegal"
<BAZ>           -->  T      ;"Sets ENV to ROOT environment"
<LEGAL? ,ENV>   -->  T      ;"ROOTs environment always legal"
```

LENGTH

```
<LENGTH structure>

MDL built-in
```

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 built-in
```

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>

MDL built-in
```

LINK links a value to PNAME str. The PNAME is placed in the specified oblist. LINK has the effect that when the MDL encounters the str it immediately replaces it with the value. LINK is primarily used in interactive mode to replace phrases that are annoyingly long to type.

Example:

```
<LINK '<INSERT-FILE "HEDGEMAZE"> "H" <ROOT>>
H    -->  Tries to load the file "HEDGEMAZE"
```

LIST

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

MDL built-in
```

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 the CONSTANT 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">>
```

Example:

```
<VERSION 5>

<LONG-WORDS? T>

<OBJECT FOO (SYNONYM HEMIDEMISEMI)>
```

```
<VOC "SUPERCALIFRAG">

<ROUTINE GO ()
     <TELL "Table length = " N <GET ,LONG-WORD-TABLE 0> CR>
     <TELL "W?SUPERCALIFRAG = " N ,W?SUPERCALIFRAG CR>
     <TELL "WORD 1 = " N <GET ,LONG-WORD-TABLE 1> CR>
     <TELL "WORD 2 = " <GET ,LONG-WORD-TABLE 2> CR>
     <TELL "W?HEMIDEMISEMI = " N ,W?HEMIDEMISEMI CR>
     <TELL "WORD 3 = " N <GET ,LONG-WORD-TABLE 3> CR>
     <TELL "WORD 4 = " <GET ,LONG-WORD-TABLE 4> CR>
>
```

LOOKUP

```
<LOOKUP string oblist>

MDL built-in
```

`LOOKUP` returns the `ATOM` with `PNAME` `string` from `oblist`. It returns `FALSE` if no `ATOM` is found.

See *The MDL Programming Language, chapter 15*.

Examples:

```
<LOOKUP "FIX" <ROOT>>              -->  FIX

FOO!-MYOBLIST
<LOOKUP "FOO" <ROOT>>              -->  #FALSE
<LOOKUP "FOO" <MOBLIST MYOBLIST>>  -->  FOO!-MYOBLIST
```

LPARSE

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

MDL built-in
```

`LPARSE` ("list parse") is just like `PARSE` with the exception that `LPARSE` returns a `LIST` of all the expressions in the `text`.

ZILF requires that the second argument is `10` if a `lookup-oblist` is given.

Examples:

```
<LPARSE "1 FOO [3]">               -->  (1 FOO [3])

<LPARSE " ">                       -->  ()

<SET A 0>
<DEFINE NXT () <SET A <+ .A 1>>>
<LPARSE "%<NXT> %<NXT> %<NXT>">    -->  (1 2 3)
```

LSH

```
<LSH number places>
```

```
MDL built-in
```

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 built-in
```

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>

ZIL library
```

`M-HPOS` returns the current horizontal cursor position on `channel`.

Example:

```
<PRINC "Hello"><M-HPOS .OUTCHAN>   -->  Hello5
```

MAKE-GVAL

```
<MAKE-GVAL atom>

ZIL library
```

MAKE-GVAL returns the atom as GVAL (,atom).

Example:

```
<SET FOO BAR>
<SETG BAR 123>
<MAKE-GVAL .FOO>              -->  ,BAR
<EVAL <MAKE-GVAL .FOO>>       -->  123
```

MAPF

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

MDL built-in
```

MAPF ("map first") traverses over all structs one element at a time until one of the structs is out of elements and calls the function applicable with the elements. In other words, the first iteration takes the first element from each of the structs and calls applicable, the second iteration takes the second element from each of the structs and calls applicable, and so on until one of the structs doesn't have any more elements. The intermediate results from each call to applicable is stored in a TUPLE.

The finalf can either be a FUNCTION or <> (FALSE). If it is FALSE the TUPLE with the intermediate result is thrown away, otherwise finalf is called with the TUPLE.

MAPF returns the result from finalf. If finalf is FALSE, MAPF returns the result from the last call to applicable. If applicable never was called (one of the structs was empty) MAPF returns FALSE.

One special case is if only finalf and applicable are given. In this case applicable is called indefinitely with no arguments until a MAPLEAVE or MAPSTOP is invoked. finalf is called if MAPSTOP is used to leave the iteration.

Examples:

```
<MAPF ,VECTOR ,+ (1 2 3) [10 11 12]>    -->  [11 13 15]

<MAPF ,STRING 1
     ["Zil" "is" "lots of" "fun"]>      -->  "Zilf"

<MAPF ,VECTOR
     <FUNCTION (N) <* .N .N>> (1 2 3)>  -->  [1 4 9]

<DEFINE SETG-MANY ("TUPLE" TUP)
     <MAPF <>
     ,SETG
     .TUP
     <REST .TUP </ <LENGTH .TUP> 2>>>>
<SETG-MANY VAR-1 VAR-2 VAR-3 100 55 616>
```

```
,VAR-1                                          -->  100
,VAR-2                                          -->  55
,VAR-3                                          -->  616

<DEFINE LNUM (N)
   <MAPF ,LIST
      <FUNCTION ()
         <COND (<=? 0 <SET N <- .N 1>>> <MAPSTOP .N>)
               (ELSE .N)>>>>
<LNUM 5>                                        -->  (4 3 2 1 0)
```

MAPLEAVE

```
<MAPLEAVE [value]>

MDL built-in
```

MAPLEAVE leaves the MAPF or the MAPR immediately and makes the MAPF or the MAPR return the value (TRUE by default). This means that an eventual finalf in the MAPF or the MAPR never will be invoked.

Example:

```
;"Return first non-zero value in STRUC"
<DEFINE FIRST-N0 (STRUC)
     <MAPF <> <FUNCTION (X)
          <COND (<N==? .X 0> <MAPLEAVE .X>)>> .STRUC>>
<FIRST-N0 [0 0 0 "ZIL" 6 0]>            -->  "ZIL"
```

MAPR

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

MDL built-in
```

MAPR ("map rest") works the same as MAPF but instead of sending one element at a time to applicable it sends the REST of the structs, starting with <REST struct 0>. In other words, the first iteration takes REST 0 from each of the structs and calls applicable, the second iteration takes REST 1 from each of the structs and calls applicable, and so on until one of the structs doesn't have any more elements. The intermediate results from each call to applicable is stored in a TUPLE.

The finalf can either be a FUNCTION or <> (FALSE). If it is FALSE the TUPLE with the intermediate result is thrown away, otherwise finalf is called with the TUPLE.

MAPR returns the result from finalf. If finalf is FALSE, MAPR returns the result from the last call to applicable. If applicable never was called (one of the structs was empty) MAPR returns FALSE.

One special case is if only finalf and applicable are given. In this case applicable is called indefinitely with no arguments until a MAPLEAVE or MAPSTOP is invoked. finalf is called if MAPSTOP is used to leave the iteration.

Example:

```
<SET FOO [1 2 3]>
;"Triple value of struct"
<MAPR <> <FUNCTION (L) <1 .L <* <1 .L> 3>>> .FOO>
.FOO                                          -->  [3 6 9]
```

MAPRET

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

MDL built-in
```

MAPRET leaves the current iteration of the MAPF or the MAPR and adds the specified values to the TUPLE of arguments used when the finalf is called. If no values are specified nothing is added to the TUPLE in this iteration. Note that the MAPF or the MAPR continues to run through the iterations until one of the structs is out of elements.

Example:

```
<SET FOO (65 66 67 68)>
<MAPF ,LIST
     #FUNCTION ((L)
     <MAPRET <ASCII .L>>) .FOO>    -->  (!\A !\B !\C !\D)
```

MAPSTOP

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

MDL built-in
```

MAPSTOP is similar to MAPRET but after it adds the values to the TUPLE of arguments it directly calls finalf and aborts all remaining iterations.

Example:

```
<DEFINE FIRST-THREE (STRUC "AUX" (I 3))
     <MAPF ,LIST
     <FUNCTION (E)
          <COND (<0? <SET I <- .I 1>>> <MAPSTOP .E>)>
     .E> .STRUC>>
<FIRST-THREE "ABCDEFG">       -->  (!\A !\B !\C)
```

MAX

```
<MAX numbers ...>

MDL built-in
```

MAX returns the maximum number among numbers.

Example:

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

MEMBER

```
<MEMBER item structure>

MDL built-in
```

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 built-in
```

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:

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

MIN

```
<MIN numbers ...>

MDL built-in
```

MIN returns the minimum number among numbers.

Example:

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

MOBLIST

```
<MOBLIST name>

MDL built-in
```

MOBLIST ("make oblist") creates and returns a new empty OBLIST named name. If an OBLIST with the name already exists the existing one is returned instead.

See *The MDL Programming Language, chapter 15*.

Example:

```
<INSERT "FOO" <MOBLIST NEW-OBLIST>>     -->  FOO!-NEW-OBLIST
FOO!-NEW-OBLIST ;"This can also be done with trailer"
```

MOD

```
<MOD number1 number2>

MDL built-in
```

`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 is 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 built-in
```

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

ZILF defines "the same object" more loosely than MDL, see `==?`.

Examples:

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

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

N=?

```
<N=? value1 value2>

MDL built-in
```

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
```

NEVER-ZAP-TO-SOURCE-DIRECTORY?

```
<NEVER-ZAP-TO-SOURCE-DIRECTORY?>

ZIL library
```

ZILF ignores this and always returns `FALSE`.

NEW-ADD-WORD

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

ZIL parser library
```

`NEW-ADD-WORD` is an alias to `ADD-WORD`.

NEWTYPE

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

MDL built-in
```

`NEWTYPE` creates a new `TYPE` with the name, `name` and the same `PRIMTYPE` as `primtype-atom`. It returns the new `TYPE`. The `name` must be unique (`<VALID-TYPE? name>` is `FALSE`> otherwise `NEWTYPE` results in an error.

It is possible to specify a `decl` (see `GDECL`) for the new `TYPE` that is enforced when `CHTYPE`.

See `APPLYTYPE`, `EVALTYPE` and `PRINTTYPE`.

Examples:

```
<NEWTYPE GARGLE CHARACTER>
<TYPEPRIM GARGLE>                       -->  FIX
<SET A <CHTYPE 65 GARGLE>>
<TYPE .A>                               -->  GARGLE
```

```
<PRIMTYPE .A>                                   -->  FIX

<NEWTYPE FIRSTNAME ATOM>
<NEWTYPE LASTNAME FIRSTNAME>
<=? ALFONSO #FIRSTNAME ALFONSO>                 -->  #FALSE
<=? #FIRSTNAME MADISON #LASTNAME MADISON>       -->  #FALSE
<=? #LASTNAME MADISON #LASTNAME MADISON>        -->  T

<NEWTYPE 2FIXLIST LIST '!<LIST FIX FIX>>
#2FIXLIST (1 2)                                 -->  Ok
#2FIXLIST (1 2 3)                               -->  Error
```

NEXT

```
<NEXT asoc>

MDL built-in
```

NEXT returns the next asoc entry, of TYPE ASOC, in the ASSOCIATION chain. If there are no more entries then FALSE is returned.

See ASSOCIATIONS, AVALUE, GETPROP, INDICATOR, ITEM and PUTPROP.

Example:

```
<DEFINE FIND-ASOC (ITEM)
     <REPEAT ((A <ASSOCIATIONS>))
          <COND (<=? .A <>> <RETURN <>>)>
          <COND (<==? .ITEM <ITEM .A>> <RETURN .A>)>
     <SET A <NEXT .A>>>>

<PUTPROP NEW-ASOC TEXT "Hello, world!">
<FIND-ASOC NEW-ASOC>
     -->  #ASOC (NEW-ASOC TEXT "Hello, world!")
```

NOT

```
<NOT value>

MDL built-in
```

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 built-in

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

OBJECT

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

ZIL library
```

OBJECT creates an object with the internal objectname, name. After the name follows LISTs of properties for the OBJECT and the values for each property. Which properties that define up a OBJECT is determined by the parser and it's possible to add new properties with PROPDEF as long as the parser is modified to support the new property. Below is a list of common properties.

IN or LOC	This is the OBJECTs initial location. This could, for example, be a ROOM, another OBJECT (container) or the player (in its inventory). There are a couple of special locations like GLOBAL-OBJECTS for OBJECTs that the player can refer to everywhere, LOCAL-GLOBALS for OBJECTs the player can refer to in ROOMs that define this OBJECT in its GLOBAL list and GENERIC-OBJECTS for OBJECTs that are concepts more than objects (for example the murder or the new will in Deadline).
SYNONYMS	This lists all the nouns that can be used to refer to the OBJECT.
ADJECTIVE	This lists all the adjectives that can be used to refer to the OBJECT.
DESC	The short description text of the OBJECT. This is the text that is, for example, printed in the players inventory.
FLAGS	This lists all the flagbits that are set on this OBJECT.
FDESC	("first description"), this is the text that is used to describe the OBJECT until it is touched (picked up).
LDESC	("long description"), this is the text that is used to describe the OBJECT, when it is on the ground, after it is touched.
GLOBAL	Optional property. This is a LIST of all the OBJECTs that is IN the LOCAL-GLOBALS that are accessible from this ROOM. This could, for example, be a door that is accessible from two different ROOMs.
THINGS	Optional property. This creates one or more simple "pseudo-objects". Each object has three parts: a LIST of adjectives (FALSE if none), a LIST of nouns and the name of the action-routine to call when this object is accessed. In early Infocom games this property was called PSEUDO and had

	a slightly different syntax.
ACTION	Defined as (ACTION routine-name). This is the OBJECTs action-routine. For OBJECTs action-routines there is no argument.
SIZE	Size of OBJECT (for inventory handling).
VALUE	Value of OBJECT (for scoring purpose).
DESCFCN	This is used to define a function to handle the OBJECTs description. It is called with an argument, ARG, that can be M-OBJDESC? or M-OBJDESC. If the routine returns FALSE during the M-OBJDESC? call, the OBJECT defaults to standard descriptions with FDESC and LDESC, otherwise the description is handled during the M-OBJDESC call.
CAPACITY	Capacity of the OBJECT if it is a container.
CONTFCN	This routine is called on the container when OBJECTs inside the container are handled (used rarely).

See *Learning ZIL, Steve E. Meretzky* and *ZIL Course, Marc S. Blank* for more on properties, flagsbits and how to write and design games.

Examples:

```
<OBJECT LAMP
     (IN LIVING-ROOM)
     (SYNONYM LAMP LANTERN LIGHT)
     (ADJECTIVE BRASS)
     (DESC "brass lantern")
     (FLAGS TAKEBIT LIGHTBIT)
     (ACTION LANTERN)
     (FDESC "A battery-powered brass lantern is
             on the trophy case.")
     (LDESC "There is a brass lantern
             (battery-powered) here.")
     (SIZE 15)>
```

OBLIST?

```
<OBLIST? atom>

MDL built-in
```

OBLIST? returns the OBLIST that contains the atom. If the atom is not in any OBLIST it returns FALSE.

See *The MDL Programming Language, chapter 15*.

Examples:

```
<==? <OBLIST? STRING> <ROOT>>                          -->  T
<OBLIST? <ATOM "SPANK-NEW-ATOM">>                      -->  #FALSE
<==? <OBLIST? FOO!-MY-OBLST> <MOBLIST MY-OBLST>>  -->  T
```

OFFSET

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

```
MDL built-in
```

OFFSET creates an OFFSET TYPE that is used with NTH and PUT to check that an element at index in the structure follows the specified pattern, structure-decl and value-decl.

The index is an integer and the structure-decl follow the normal rules for a decl (see GDECL). Because the OFFSET only specifies the decl for one element in the structure it is possible to split the decl in two parts where structure-decl specifies the structure and value-decl is the decl for this specific element.

Note that in ZILF can OFFSET only be used with NTH and PUT in the form <index-or-offset structure> and <index-or-offset structure value> respectively.

GET-DECL and PUT-DECL can be used to examine and change the decl of the OFFSET and INDEX returns the index of an OFFSET.

Example:

```
<SETG OFF1 <OFFSET 1 '<VECTOR FIX>>>
<SETG OFF2 <OFFSET 2 '<VECTOR FIX CHARACTER>>>
<SETG OFF3 <OFFSET 3 '<VECTOR> 'STRING>>
<GET-DECL ,OFF2>              -->  <VECTOR FIX CHARACTER>
<SET V [1 !\A "BCD"]>
<OFF1 .V>                     -->  1
<OFF3 .V>                     -->  "BCD"
<OFF2 .V !\B>                 -->  [1 !\B "BCD"]
<OFF1 .V !\A>                 -->  ERROR
<2 .V 65>
<OFF2 .V>                     -->  ERROR
```

OPEN

```
<OPEN "READ" path>

MDL built-in
```

OPEN the file at path for input. The second argument must always be "READ" in ZILF and the path is specified like in Linux (forward slashes etc.) and uppercase/lowercase can be significant, depending on the host system.

Example:

```
;"ZILF ver 0.9"
<SET CH <OPEN "READ" "../zillib/parser.zil">>
<SET BUFFER <ISTRING 1000>>
<READSTRING .BUFFER .CH ";">  -->  124 ;"READ until first ;"
<CLOSE .CH>
```

OR

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

```
MDL built-in
```

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 built-in
```

Returns the same result as OR with the difference that all exressions 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 built-in
```

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 OBJECTdefinition.

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>

MDL package system
```

PACKAGE defines a group of ATOMs (i.e. variables and functions) with the package-name for potential later inclusion (via USE or USE-WHEN) in the project. A PACKAGE is often used to functionally group together library functions that can have a usage over many projects.

Internally an OBLIST named PACKAGE is used in conjunction with BLOCK and ENDBLOCK. When you define a PACKAGE the following is happening:

1. An external OBLIST, package-name, is created and added to the OBLIST PACKAGE (e.g. FOO!-PACKAGE).
2. An internal OBLIST, Ipackage-name, is created and added to the OBLIST package-name (e.g. IFOO!-FOO!-PACKAGE).
3. A BLOCK is started with the OBLISTs (in this order) Ipackage-name, package-name and <ROOT> (e.g. IFOO, FOO, <ROOT>).

This means that every ATOM that is created inside the PACKAGE ends up on the internal OBLIST first. If ENTRY is used the ATOM is created/moved to the external OBLIST and finally RENTRY creates/moves the ATOM to the ROOT OBLIST.

The PACKAGE definition is ended by END-PACKAGE (in fact an ENDBLOCK) which restores the OBLISTs to the state they had before the PACKAGE definition began.

When you decide to use a package by USE or USE-WHEN the OBLIST package-name is copied and added last to the local OBLIST (<LVAL OBLIST>). This means that all ATOMs on the external package OBLIST becomes available in current environment.

Note that a PACKAGE can be defined additive (i.e. multiple PACKAGE definitions with the same package-name is added together to one PACKAGE).

ZILF has three packages predefined in <MOBLIST PACKAGE>; NEWSTRUC, ZIL and ZILCH. They are all empty and are only there for compatibility (all ATOMs in these packages are already in ZILF).

See DEFINITIONS, ENDPACKAGE, ENTRY, RENTRY, USE and USE-WHEN.

Examples:

```
;"Define PACKAGE"
```

```
<REMOVE ANSWER> ;"Secure that ATOM not on any OBLIST"
<REMOVE DBL-ANSWER>
<REMOVE ROOT-ANSWER>
<REMOVE SECRET>
<PACKAGE "FOO">
<ENTRY ANSWER>
<SETG ANSWER 42>
<SETG SECRET 12345>
<RENTRY ROOT-ANSWER>
<SETG ROOT-ANSWER 21>
<ENDPACKAGE>

<TYPE? <GETPROP FOO!-PACKAGE OBLIST> OBLIST>      -->  OBLIST
<TYPE? <GETPROP IFOO!-FOO!-PACKAGE OBLIST> OBLIST>-->  OBLIST
<GASSIGNED? ANSWER>                               -->  #FALSE
<GASSIGNED? ANSWER!-FOO!-PACKAGE>                 -->  T
<GASSIGNED? SECRET!-IFOO!-FOO!-PACKAGE>           -->  T
,ANSWER!-FOO!-PACKAGE                             -->  42
,SECRET!-IFOO!-FOO!-PACKAGE                       -->  12345
,ROOT-ANSWER                                      -->  21

;"PACKAGEs can be defined additive"
<PACKAGE "FOO">
<SETG DBL-ANSWER <* ,ANSWER 2>>
<ENTRY DBL-ANSWER>
<ENDPACKAGE>

,ANSWER!-FOO!-PACKAGE                             -->  42
,DBL-ANSWER!-FOO!-PACKAGE                         -->  84

;"USE adds external OBLIST to local OBLIST-path"
<REMOVE ANSWER> ;"Secure that ATOM not on any OBLIST"
<LENGTH .OBLIST>                                  -->  2
<USE "FOO">
<LENGTH .OBLIST>                                  -->  3
,ANSWER                                           -->  42
<GASSIGNED? SECRET>                               -->  #FALSE
,SECRET!-IFOO                                     -->  12345
```

PARSE

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

MDL built-in
```

`PARSE` takes a string, `text`, and returns the first MDL object encountered in it. If `lookup-oblist` is supplied, `PARSE` looks for potential `ATOM`s on this `OBLIST`. If no `lookup-oblist` is supplied, `.OBLIST` is used.

ZILF requires that the second argument is `10` if a `lookup-oblist` is supplied.

Examples:

```
<PARSE "FOO">                                --> FOO

<PARSE "+">                                  --> +
<PARSE "+" 10 <GETPROP PACKAGE OBLIST>> --> +!-PACKAGE

<PARSE "23">                                 --> 23

<PARSE "(1 2 3)">                            --> (1 2 3)

<PARSE "<+ 12 34>">                          --> <+ 12 34>
<PARSE "%<+ 12 34>">                         --> 46

<PARSE "<+ .A .B>" 10 <MOBLIST OB>>
          --> <+!-OB <LVAL!-OB A!-OB> <LVAL!-OB B!-OB>>

<PARSE " ">                     --> ERROR (No expression)

<PARSE "1 2 3">                 --> 1 (Only 1st expression)
```

PICFILE

```
<PICFILE>

ZIL library
```

ZILF ignores this and always returns `FALSE`.

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

PNAME

```
<PNAME atom>

MDL built-in
```

`PNAME` ("printed name") returns a newly created string copy of the `atom`'s pname. `PNAME` never prints an `ATOM`s trailers, unlike `UNPARSE`, and is therefore quicker.

See *The MDL Programming Language, chapter 15*.

Examples:

```
<PNAME FOO>                     --> "FOO"

<PNAME FOO!-NEW-OBLIST>         --> "FOO"
<UNPARSE FOO!-NEW-OBLIST>       --> "FOO!-NEW-OBLIST"
```

PREP-SYNONYM

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

ZIL parser library
```

PREP-SYNONYM creates one or more synonyms to the original preposition.

ZILF treats PREP-SYNONYM as an alias to SYNONYM.

PRIMTYPE

```
<PRIMTYPE value>

MDL built-in
```

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 built-in
```

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 built-in
```

PRINC is just 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 built-in
```

PRINT is just like PRIN1, 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 ...>

ZIL library
```

PRINT-MANY prints multiple items to channel with the printer. The printer is usually PRINT, PRINC or PRIN1 but could actually be any FUNCTION that takes one argument. The printer is called repeatedly with one item at a time until the list of items is exhausted.

If PRMANY-CRLF is given as an item, a CRLF is printed at that position.

Examples:

```
<PRINT-MANY .OUTCHAN PRINC "Hello" !\! PRMANY-CRLF>
     -->  Hello!\n

<PRINT-MANY .OUTCHAN PRIN1 "string" !\c PRMANY-CRLF>
     -->  "string"!\c\n
```

PRINTTYPE

```
<PRINTTYPE atom [handler]>

MDL built-in
```

PRINTTYPE tells the TYPE atom how it should be printed (PRIN1-style). If PRINTTYPE is called without a handler then the currently active handler is returned. If there is no active handler, FALSE is returned.

Note that it is possible to replace the handler with a new handler, even on the predefined TYPEs.

See APPLYTYPE, EVALTYPE and NEWTYPE.

Examples:

```
<DEFINE ROMAN-PRINT (ROMAN "AUX" (RNUM <CHTYPE .ROMAN FIX>))
<COND (<OR <L=? .RNUM 0> <G? .RNUM 3999>>
       <PRINC <CHTYPE .NUMB TIME>>)
      (T
       <RCPRINT </ .RNUM 1000> '![!\M]>
       <RCPRINT </ .RNUM 100>  '![!\C !\D !\M]>
       <RCPRINT </ .RNUM 10>   '![!\X !\L !\C]>
       <RCPRINT    .RNUM       '![!\I !\V !\X]>)>>

<DEFINE RCPRINT (MODN V)
<SET MODN <MOD .MODN 10>>
<COND (<==? 0 .MODN>)
       (<==? 1 .MODN> <PRINC <1 .V>>)
       (<==? 2 .MODN> <PRINC <1 .V>> <PRINC <1 .V>>)
       (<==? 3 .MODN> <PRINC <1 .V>> <PRINC <1 .V>>
                                     <PRINC <1 .V>>)
       (<==? 4 .MODN> <PRINC <1 .V>> <PRINC <2 .V>>)
       (<==? 5 .MODN> <PRINC <2 .V>>)
       (<==? 6 .MODN> <PRINC <2 .V>> <PRINC <1 .V>>)
       (<==? 7 .MODN> <PRINC <2 .V>> <PRINC <1 .V>>
                                     <PRINC <1 .V>>)
       (<==? 8 .MODN> <PRINC <2 .V>> <PRINC <1 .V>>
                      <PRINC <1 .V>> <PRINC <1 .V>>)
       (<==? 9 .MODN> <PRINC <1 .V>> <PRINC <3 .V>>)>>

<NEWTYPE ROMAN FIX>
<PRINTTYPE ROMAN ,ROMAN-PRINT>
<==? <PRINTTYPE ROMAN> ,ROMAN-PRINT>
#ROMAN 1984                              -->  MCMLXXXIV

<NEWTYPE ROMAN2 FIX>
<PRINTTYPE ROMAN2 ROMAN> ;"Copies active handler, if exists"
#ROMAN2 2020                             -->  MMXX

<PRINTTYPE ROMAN FIX>
<=? <PRINTTYPE ROMAN> <>>                -->  T
#ROMAN 2020                              -->  2020
;"Change in ROMAN doesn't affect ROMAN2"
#ROMAN2 2020                             -->  MMXX

<PRINTTYPE FIX ,ROMAN-PRINT> ;"Works on built-in too!"
23                                       -->  XXIII

<PRINTTYPE FORM <FUNCTION (F) <PRIN1 <CHTYPE .F LIST>>>>
<FORM + 1 2>                             -->  (+ I II)
```

PROG

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

MDL built-in

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 GDECL.

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-patterns ...]>

ZIL library
```

PROPDEF defines a property, atom, with a default-value for OBJECTs (and ROOMs). The default-value is the value that GETP will return if the property is not defined for the given OBJECT.

For the more complex properties it is possible to define a spec-pattern according to:

```
(atom|DIR ["MANY"|"OPT"] [phrase] var:type ... =
          [form-len] ["MANY"] <fnc-size var>|(const value)|
          (ptr <fnc-size var>) ...) ...
```

The spec-pattern consists of two parts divided by an equal sign. The left side is the pattern and the right side is the rules on how to store the property.

atom|DIR	This is the property. DIR is a special case that is used for DIRECTIONS.
"MANY"	This means that the pattern of var:type repeats itself. If "MANY" is defined on the left side of the equal sign there must be a matching

	on the right side.
"OPT"	This means that the pattern after is optional.
[phrase]	This can be tokens like IF, ELSE, TO.
var:type	This is a variable name, var, and its type. Usually FIX, STRING or ROOM.
form-len	The length (records) of the form. The form-len is optional and can also be given as <>.
<fnc-size var>	The fnc-size can be a call with var to either BYTE, WORD, STRING, OBJECT, ROOM, GLOBAL, NOUN, ADJ, or VOC. This stores var or derivative of var and adds to the vocabulary and/or creates a GVAL.
(ptr <fnc-size var>)	This creates a GVAL, ptr, that contains the address-pointer relative to the property.
(const value)	This creates a CONSTANT, name, containing value.

Examples:

```
;"Ordinary property"
<PROPDEF HEIGHT 72>
<OBJECT OBJ1>
<OBJECT OBJ2 (HEIGHT 80)>
;"Implies, inside routine"
<GETP ,OBJ1 ,P?HEIGHT>              -->  72
<GETP ,OBJ2 ,P?HEIGHT>              -->  80

;"Basic pattern"
<PROPDEF HEIGHT <>
     (HEIGHT FEET:FIX FOOT INCHES:FIX = 2 <WORD .FEET>
                                          <BYTE .INCHES>)
     (HEIGHT FEET:FIX FT INCHES:FIX = 2 <WORD .FEET>
                                        <BYTE .INCHES>)>
<OBJECT GIANT (HEIGHT 10 FT 8)>
;"Implies, inside routine"
<=? <GET <GETPT ,GIANT ,P?HEIGHT> 0> 10>
<=? <GETB <GETPT ,GIANT ,P?HEIGHT> 2> 8>

;"Basic pattern with OPT"
<PROPDEF HEIGHT <> (HEIGHT FEET:FIX FT "OPT" INCHES:FIX =
                <WORD .FEET> <BYTE .INCHES>)>
<OBJECT GIANT1 (HEIGHT 100 FT)>
<OBJECT GIANT2 (HEIGHT 50 FT 11)>
;"Implies, inside routine"
<=? <PTSIZE <GETPT ,GIANT1 ,P?HEIGHT>> 3>
<=? <GET <GETPT ,GIANT1 ,P?HEIGHT> 0> 100>
<=? <GETB <GETPT ,GIANT1 ,P?HEIGHT> 2> 0>
<=? <PTSIZE <GETPT ,GIANT2 ,P?HEIGHT>> 3>
<=? <GET <GETPT ,GIANT2 ,P?HEIGHT> 0> 50>
<=? <GETB <GETPT ,GIANT2 ,P?HEIGHT> 2> 11>

;"Basic pattern with MANY"
<PROPDEF TRANSLATE <> (TRANSLATE "MANY" A:ATOM N:FIX =
                "MANY" <VOC .A BUZZ> <WORD .N>)>
```

```
<OBJECT NUMBERS (TRANSLATE ONE 1 TWO 2)>
;"Implies, inside routine"
<=? <PTSIZE <GETPT ,NUMBERS ,P?TRANSLATE>> 8>
<=? <GET <GETPT ,NUMBERS ,P?TRANSLATE> 0> ,W?ONE>
<=? <GET <GETPT ,NUMBERS ,P?TRANSLATE> 1> 1>
<=? <GET <GETPT ,NUMBERS ,P?TRANSLATE> 2> ,W?TWO>
<=? <GET <GETPT ,NUMBERS ,P?TRANSLATE> 3> 2>

;"Pattern with constants"
<PROPDEF HEIGHT <> (HEIGHT FEET:FIX FT INCHES:FIX =
     (HEIGHTSIZE 3) (H-FEET <WORD .FEET>)
     (H-INCHES <BYTE .INCHES>))>
<=? ,HEIGHTSIZE 3>
<=? ,H-FEET 0>
<=? ,H-INCHES 2>

;"DIR sets pattern for all DIRECTIONS"
<PROPDEF DIRECTIONS <> (DIR GOES TO R:ROOM =
     (MY-UEXIT 3) <WORD 0> (MY-REXIT <ROOM .R>))>
<DIRECTIONS NORTH SOUTH>
<OBJECT HOUSE (SOUTH GOES TO WOODS)>
<OBJECT WOODS (NORTH GOES TO HOUSE)>
;"Implies, inside routine"
<=? <PTSIZE <GETPT ,HOUSE ,P?SOUTH>> ,MY-UEXIT>
<=? <GETB <GETPT ,HOUSE ,P?SOUTH> ,MY-REXIT> ,WOODS>

;"DIR sets implicit DIRECTIONS"
<PROPDEF DIRECTIONS <> (DIR GOES TO R:ROOM =
     (MY-UEXIT 3) <WORD 0> (MY-REXIT <ROOM .R>))>
<DIRECTIONS NORTH SOUTH>
<OBJECT HOUSE (EAST GOES TO WOODS)>
<OBJECT WOODS (WEST GOES TO HOUSE)>
;"Implies, inside routine"
<=? <PTSIZE <GETPT ,HOUSE ,P?EAST>> ,MY-UEXIT>
<=? <GETB <GETPT ,HOUSE ,P?EAST> ,MY-REXIT> ,WOODS>
<BAND <GETB ,W?EAST 4> ,PS?DIRECTION>

;"VOC in pattern adds word to vocabulary"
<PROPDEF FOO <> (FOO A:ATOM = <VOC .A PREP>)>
<OBJECT BAR (FOO FOO)>
;"Implies, inside routine"
<=? <GETP ,BAR ,P?FOO> ,W?FOO>

;"Complex PROPDEF (DIRECTIONS from Zork Zero)"
<PROPDEF DIRECTIONS <>
     (DIR TO R:ROOM = (UEXIT 1) (REXIT <ROOM .R>))
     (DIR S:STRING = (NEXIT 2) (NEXITSTR <STRING .S>))
     (DIR SORRY S:STRING = (NEXIT 2) (NEXITSTR <STRING .S>))
     (DIR PER F:FCN = (FEXIT 3)
                      (FEXITFCN <WORD .F>) <BYTE 0>)
     (DIR TO R:ROOM IF F:GLOBAL "OPT" ELSE S:STRING =
          (CEXIT 4) (REXIT <ROOM .R>) (CEXITFLAG <GLOBAL .F>)
```

```
                                        (CEXITSTR <STRING .S>))
        (DIR TO R:ROOM IF O:OBJECT IS OPEN "OPT" ELSE S:STRING =
             (DEXIT 5) (DEXITOBJ <OBJECT .O>)
             (DEXITSTR <STRING .S>) (DEXITRM <ROOM .R>))>
```

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 built-in
```

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>

MDL built-in
```

`PUT-DECL` defines an alias, `item`, for a `pattern`. See `DECL?`, `GDECL` and `GET-DECL` for more on declaration patterns.

Examples:

```
<DECL? T BOOLEAN>                       -->  Error

<PUT-DECL BOOLEAN '<OR ATOM FALSE>>
<DECL? T BOOLEAN>                       -->  T
<DECL? "Hi" BOOLEAN>                    -->  #FALSE
```

PUT-PURE-HERE

```
<PUT-PURE-HERE>
```

```
ZIL library
```

ZILF ignores this and always returns FALSE.

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 built-in 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]>

MDL built-in
```

PUTPROP stores value as an association on the item under the indicator and returns the item. If no value is specified PUTPROP returns the value and then clears the association.

In ZILF there is a special indicator, PROPSPEC, that has a special meaning inside OBJECTs. A PROPSPEC property is defined:

```
<PUTPROP item PROPSPEC [function]>
```

When an item defined in this way is used in an OBJECT, the function is invoked during the compilation with the LIST (containing the item) as an argument. The return value from the function must be a LIST and it is stored as value under PROPSPEC on the item. If no function is specified the PROPSPEC for the item is cleared. See examples below.

See ASSOCIATIONS, AVALUE, GETPROP, INDICATOR, ITEM and NEXT.

Examples:

```
<SET L (1 2 3)>
<PUTPROP .L FOO "Hello">       -->  (1 2 3)
<GETPROP .L FOO>               -->  "Hello"
<PUTPROP .L FOO>               -->  "Hello"
<GETPROP .L FOO>               -->  #FALSE

;"PROPSPEC, loop through all words and add to buzz"
<VERSION XZIP>
<OBJECT FOO
```

```
     (ADJECTIVE SMALL CURIOUS)
     (MYBUZZ "ABCD" "BAR" "BAZ")>
<DEFINE MYBUZZ-PROP (L)
     <SET L <REST .L>>   ;"Ignore MYBUZZ in LIST"
     <MAPF ,LIST <FUNCTION (W) <VOC .W BUZZ>> .L>>
<PUTPROP MYBUZZ PROPSPEC MYBUZZ-PROP>
<ROUTINE GO () <TEST-PROPSPEC>>
<ROUTINE TEST-PROPSPEC ("AUX" W)
     <TELL "Part-of-Speech, 4 = BUZZ" CR>
     <SET W W?ABCD>
     <TELL "ABCD = " N <GETB .W 6> CR>
     <SET W W?BAR>
     <TELL "BAR = " N <GETB .W 6> CR>
     <SET W W?BAZ>
     <TELL "BAZ = " N <GETB .W 6> CR>>
```

PUTREST

```
<PUTREST list new-rest>

MDL built-in
```

`PUTREST` replaces the `REST` of `list` with `new-rest` and returns `list`. In other words, `list` is assigned the first element of `list` and then all the elements from `new-rest`. Note that this actually changes the `list`.

Examples:

```
<PUTREST (1 2 3) (A B)>       -->  (1 A B)

<SET L1 [<SET L2 (1 2 3)>]>
<PUTREST .L2 (A B)>
.L1                           -->  [(1 A B)]

<SET L1 [1 2 3]>
<SET L2 <PUTREST (!.L1) (A B)>>
.L1                           -->  [1 2 3]
.L2                           -->  (1 A B)

<SET L1 (1 2 3 4 5 6 7 8 9)>
<PUTREST <REST .L1 3> <REST .L1 7>>
.L1                           -->  (1 2 3 4 8 9)
```

QUIT

```
<QUIT [exit-code]>

MDL built-in
```

`QUIT` exits ZILF (interpreter mode) and returns to the operating system with `exit-code`.

Example:

```
<QUIT>
```

QUOTE

```
<QUOTE value>
'value                          ;"Alternative syntax"
MDL built-in
```

QUOTE returns value unevaluated.

Examples:

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

READSTRING

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

MDL built-in
```

READSTRING reads bytes from the channel into buffer-str and returns the number of bytes read into buffer-str. The buffer-str needs to have room for the input. For each call to READSTRING it either reads bytes to fill up the buffer-str or until max-length-or-stop-chars is reached. The max-length-or-stop-chars can be a FIX number of bytes or a STRING that halts input.

READSTRING returns the actual number of bytes read and returns 0 when the EOF is reached.

Example:

```
;"ZILF ver 0.9"
<SET CH <OPEN "READ" "../zillib/parser.zil">>
<SET BUFFER <ISTRING 10>>
<READSTRING .BUFFER .CH>           -->  10
<LVAL BUFFER>                      -->  "\"Library h"
<READSTRING .BUFFER .CH 6>         -->  6
<LVAL BUFFER>                      -->  "eader\"ry h"
<READSTRING .BUFFER .CH "ZIL">     -->  10
<LVAL BUFFER>                      -->  "\n\n<SETG "
<CLOSE .CH>                             ;"\n = CR+LF"
```

REMOVE

```
<REMOVE pname oblist>
<REMOVE atom>

MDL built-in
```

This REMOVEs the ATOM with pname from oblist. It returns FALSE If the ATOM is not on the

oblist.

<REMOVE atom> REMOVEs the atom from its OBLIST. FALSE is returned if it's not on its OBLIST.

See *The MDL Programming Language, chapter 15*.

Examples:

```
FOO
<1 .OBLIST>                 -->  (... ("FOO" FOO))
<REMOVE FOO>
<1 .OBLIST>                 -->  FOO is removed from <1 .OBLIST>

FOO-1!-OB
FOO-2!-OB
<MOBLIST OB>                    -->  FOO-1, FOO-2 on OB
<REMOVE "FOO-1" <MOBLIST OB>> -->  FOO-1!-#FALSE ()
<MOBLIST OB>                    -->  Only FOO-1 on OB
<REMOVE FOO-2!-OB>
<MOBLIST OB>                    -->  OB is empty
```

RENTRY

```
<RENTRY atoms ...>

MDL package system
```

RENTRY creates/moves one or more ATOMs to <ROOT> in a PACKAGE or DEFINITION. RENTRY is only valid inside a PACKAGE or DEFINITION, if it's used outside an error is raised.

See DEFINITIONS, ENTRY, INCLUDE, INCLUDE-WHEN, PACKAGE, USE, USE-WHEN.

Examples:

```
<REMOVE ANSWER> ;"Secure that ATOM not on any OBLIST"
<PACKAGE "FOO">
<SETG ANSWER 42>
<RENTRY ANSWER>
<ENDPACKAGE>

,ANSWER         -->  42 ;"Accessible without previous USE"
```

REPEAT

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

MDL built-in
```

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 GDECL.

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") where 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 and REPLACE-DEFINITION..

REST

```
<REST structure [count]>

MDL natvive
```

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 built-in
```

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 ...) ...>

ZIL library
```

ROOM creates a room-object with the internal objectname, name. After the name follows LISTs of properties for the ROOM and the values for each property. Which properties that define up a ROOM is determined by the parser and it's possible to add new properties with PROPDEF as long as the parser is modified to support the new property. Usually the below properties are understood by the parser and the properties IN (or LOC), DESC and FLAGS are required, the others are optional.

IN or LOC	Required property. The value is always ROOMS for ROOM-objects.
DESC	Required property. The short description text of the ROOM. This is the text that is, for example, printed in the statusbar.
FLAGS	Required property. This lists all the flagbits that are set on this ROOM.
LDESC	Optional property. The long description of the ROOM. This is the text that is printed, for example, the first time the player visits the ROOM
(dir ...)	Optional property. List a direction, dir and where it leads. There is 5

different types of EXITS:
UEXIT ("unconditional exit"). The syntax is `(dir TO room-name)`. If the player moves in this direction he is moved unconditionally to `room-name`.
NEXIT ("non-exit"). The syntax is `(dir "text-why-not")`. The `text-why-not` is printed when the player tries to move in this direction. Use this only if you want a different text than the standard message, typically something like `"You can't move in that direction!"`.
CEXIT ("conditional exit"). The syntax is `(dir TO room-name IF gval [ELSE "text-why-not"])`. This moves the player if the global value, `gval`, is `TRUE`. The `ELSE`-part is optional and the standard message is printed if it is not supplied.
DEXIT ("door-exit"). The syntax is `(dir TO room-name IF door-name IS OPEN)`. This is a special case of `CEXIT` that moves the player to `room-name` if the `door-name` has the `OPENBIT` set.
FEXIT ("function-exit"). The syntax is `(dir PER routine-name)`. This moves the player to the `ROOM` returned by the `ROUTINE`, `routine-name`. If the routine returns `FALSE` it is presumed that the routine has printed an appropriate message.

`GLOBAL` Optional `property`. This is a `LIST` of all the `OBJECT`s that is `IN` the `LOCAL-GLOBALS` that are accessible from this `ROOM`. This could, for example, be a door that is accessible from two different `ROOM`s.

`THINGS` Optional `property`. This creates one or more simple "pseudo-objects". Each object has three parts: a `LIST` of adjectives (`FALSE` if none), a `LIST` of nouns and the name of the action-routine to call when this object is accessed. In early Infocom games this property was called `PSEUDO` and had a slightly different syntax.

`ACTION` Optional `property`. The syntax is `(ACTION routine-name)`. This `ROUTINE` takes one argument, by convention call `RARG` ("room-argument"), and is called more than once during a turn with different values to `RARG`.
`M-BEG`, the `routine-name` is called with this value to `RARG` before any `OBJECT`s or `verb` action-routines.
`M-END`, the `routine-name` is called with this value to `RARG` after any `OBJECT`s or `verb` action-routines.
`M-LOOK`, the `routine-name` is called with this value to `RARG` when the player `LOOK`s.
`M-ENTER`, the `routine-name` is called with this value to `RARG` when the player enters the `ROOM` (before any room description).

Note that `ROOM`s can just as easily be created with `OBJECT` as long as they are `(IN ROOMS)`.

See *Learning ZIL, Steve E. Meretzky* and *ZIL Course, Marc S. Blank* for more on properties, flagsbits and how to write and design games.

Example:

```
<ROOM INSIDE-HOUSE
    (DESC "Inside House")
    (IN ROOMS)
    (LDESC
"You are standing inside the rotting house. The house is
```

```
 sparsely furnished, in fact not at all. On one wall is
 positioned a sign. Beside the sign is a button, and an open
 trap-door is placed on the floor. The exit is west
 and there is a walk-in closet in the eastern wall.")
(UP "You have yet to master the art of flying.")
(EAST TO CLOSET)
(WEST TO OUTSIDE-HOUSE IF FRONT-DOOR-FLAG ELSE ,MSG-025)
(DOWN PER TRAP-DOOR-F)
(ACTION INSIDE-HOUSE-F)
(FLAGS LIGHTBIT NDUNGEONBIT)
(THINGS (<>) (BUTTON) LIGHTBUTTON-F
        (<>) (SIGN) HOUSE-SIGN-F
        (<>) (HOUSE FLOOR CLOSET KEYHOLE) STANDARD-F)
(GLOBAL FRONT-DOOR)>
```

ROOT

```
<ROOT>

MDL built-in
```

ROOT returns the OBLIST containing names of primitives (the same as <2 .OBLIST>). Initially it contains all predefined SUBRs or FSUBRs, as well as OBLIST, DEFAULT, T, etc.

See *The MDL Programming Language, chapter 15*.

ROUTINE

```
<ROUTINE name [activation-atom] arg-list body ...>

ZIL library
```

The ROUTINEs are the central building block in a ZIL-program. Inside the ROUTINE it is only possible to use the reduced instruction set that can be executed on the Z-machine. It is the instructions inside the ROUTINEs that are compiled to the actual ZIP-program.

ROUTINE defines a program block with its own set of bindings. It is possible to specify an activation-atom to use as an argument to control the RETURN statement inside the ROUTINE.

The arg-list is formatted the same way as FUNCTION, but the legal tokens is reduced to these:

Arguments	The required arguments for this ROUTINE. The arguments are bound to local variables inside this ROUTINE.
"OPT"	The optional arguments for this ROUTINE. The arguments are bound to local variables inside this ROUTINE and can be defined with a default value. "OPTIONAL" is an alias for "OPT".
"AUX"	Followed by any number of ATOMs that becomes local variables inside this ROUTINE and can be defined with a default value. "EXTRA" is a alias for "AUX".
"NAME"	Followed by an ATOM that becomes the activation-atom for this ROUTINE. This is equivalent to naming the activation-atom before

the arg-list. "ACT" is an alias for "NAME".

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.

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

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

After the arg-list follows the ZIL-instructions that makes up the body of the ROUTINE.

Example:

```
;"Move all child objects from object src to object dst"
<ROUTINE MOVE-INVENTORY (SRC DST "AUX" X N)
     <SET X <FIRST? .SRC>>
     <REPEAT ()
          <COND (.X
               <SET N <NEXT? .X>>
               <MOVE .X .DST>
               <SET X .N>)
          (T <RETURN>)>>>
```

ROUTINE-FLAGS

```
<ROUTINE-FLAGS CLEAN-STACK?>

ZIL library
```

This sets flags to control how ZILF should compile. To clear, call FILE-FLAGS without any flags. The flags are:

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.

Examples:

```
<FILE-FLAGS CLEAN-STACK?>
```

SET

```
<SET atom value [environment]>

MDL built-in
```

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 ...>

MDL built-in
```

SET-DEFSTRUCT-FILE-DEFAULTS is used to change the default behaviour of the struct-option and the field-option tokens in DEFSTRUCT.

The newly defined defaults are only active in the same file as they were defined. If a file is loaded via, for example, FLOAD or INSERT-FILE the defaults are the built-in defaults inside these files.

If SET-DEFSTRUCT-FILE-DEFAULTS is called without any arguments the built-in default behaviour is restored.

The tokens that can have changed default behaviour are:

'CONSTRUCTOR	Replace the default constructor (MAKE-).
'INIT-ARGS	Replace the init arguments to the base-type. This is empty by default.
'NODECL	Use 'NODECL, to get 'NODECL by default.
'NOTYPE	Use 'NOTYPE, to get 'NOTYPE by default.
'NTH	Default ATOM for this is NTH. Change to other with ('NTH MY-NTH).
'PRINTTYPE	Change the default ATOM for PRINTTYPE with ('PRINTTYPE MY-PRINTTYPE).
'PUT	Default ATOM for this is PUT. Change to other with ('PUT MY-PUT).
'START-OFFSET	Default value is 1. Change with ('START-OFFSET value).

See DEFSTRUCT for more on user defined structures.

Example:

```
<SET-DEFSTRUCT-FILE-DEFAULTS ('NTH GETB) ('PUT PUTB)
     ('START-OFFSET 0) 'NODECL ('INIT-ARGS (BYTE))>
<DEFSTRUCT B-TBL TABLE (B-TBL-X FIX 65) (B-TBL-Y FIX 111)>
<MAKE-B-TBL>              -->  #B-TBL %<TABLE (BYTE) 65 111>
<B-TBL-Y <MAKE-B-TBL>>    -->  111
```

SETG

```
<SETG atom value>

MDL built-in
```

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]  ...]>

MDL built-in
```

SORT can sort a VECTOR (or TUPLE). The predicate can either be <> or a FUNCTION that takes two keys and returns TRUE if the two records are correctly sorted and FALSE if they are incorrectly sorted. For example ,G? will sort keys in ascending order and ,L? will sort keys in descending order. If the predicate is <> the keys must be of the same TYPE and the vector will be sorted in ascending order.

The record-size is the length of each record (default value is 1) and the key-offset is the offset in the record to the value to use as the sort key (default value is 0).

If additional vectors are supplied all vectors can have their own record length but each vector must have the same number of records. Records in the additional vectors are interchanged based on how the main vector is sorted.

SORT returns the first sorted vector.

Examples:

```
<SORT <> [3 4 2 1]>      -->  [1 2 3 4]

<SET V [1 MONEY 2 SHOW 3 READY 4 GO]>
<SORT <> .V 2 1>         -->  [4 GO 1 MONEY 3 READY 2 SHOW]
<SORT ,L? .V 2>          -->  [4 GO 3 READY 2 SHOW 1 MONEY]

<SET V [1 MONEY 2 SHOW 3 READY 4 GO]>
<SORT <> [5 1 6 3 7 2 8 4] 1 0 .V 1>
.V                       -->  [MONEY READY SHOW GO 1 2 3 4]
```

SPNAME

```
<SPNAME atom>

MDL built-in
```

SPNAME ("shared printed name") should return the same string of the atom's pname that is in its OBLIST (i.e. pointing to the same storage and therefore not able to change or modify).

ZILF treats SPNAME as an alias to PNAME and returns a string copy of the atom's pname.

See PNAME and *The MDL Programming Language, chapter 15*.

STRING

```
<STRING values ...>

MDL built-in
```

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 built-in
```

STRUCTURED? is a predicate and returns true if value is of a structured TYPE. The structured TYPEs are:

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

Examples:

```
<STRUCTURED? <LIST 1 2 3>>    -->  T
<STRUCTURED? <TABLE 1 2 3>>   -->  #FALSE
```

SUBSTRUC

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

MDL built-in
```

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 = "BCEEEEEE"
```

SUPPRESS-WARNINGS?

```
<SUPPRESS-WARNINGS? all | none | codes ...>

ZILF compiler directive
```

SUPPRESS-WARNINGS? tells the compiler how to treat warnings. NONE is the default.

ALL	Suppress all warnings.
NONE	Don't suppress any warnings.
codes	Suppress listet warning codes.

Examples:

```
;"Examples must be compiled with -w, otherwise warnings is
  always suppressed."

;"Compiles with warnings"
<SUPPRESS-WARNINGS? NONE>
<GLOBAL X 5>
<ROUTINE GO () <TELL N .X>>

;"Compiles with suppressed warnings"
<SUPPRESS-WARNINGS? ALL>
<GLOBAL X 5>
<ROUTINE GO () <TELL N .X>>

;"Compiles with suppressed warnings"
<SUPPRESS-WARNINGS? "ZIL0204">
<GLOBAL X 5>
<ROUTINE GO () <TELL N .X>>
```

SYNONYM

```
<SYNONYM original synonyms ...>

ZIL parser library
```

SYNONYM creates one or more synonyms to the original verb, adjective, preposition or direction. Instead of SYNONYM it is also possible to use VERB-SYNONYM, ADJ-SYNONYM, PREP-SYNONYM and DIR-SYNONYM for verbs, adjectives, prepositions and directions respectively, ZILF handles them all like aliases to SYNONYM.

Note that due to the way words, especially adjectives and nouns, are stored in the vocabulary synonyms for adjectives only work in version 3 (ZIP) games.

Examples:

```
<SYNONYM NORTH FORE>
<SYNONYM SOUTH AFT>
<SYNONYM WEST PORT>
<SYNONYM EAST STARBOARD>

<SYNTAX PUT OBJECT = V-INSERT>
<VERB-SYNONYM PUT SLIDE DIP SOAK>
```

SYNTAX

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

ZIL parser library
```

`SYNTAX` defines a `verb`-phrase and specifies which `action-routine-name` should be called when an input matches this `verb`-phrase. A `SYNTAX` must contain a `verb` and an `action-routine-name`. Optionally it can contain one direct noun-phrase, the first token `OBJECT`, and one indirect noun-phrase, the second token `OBJECT`. Each noun-phrase can also have a corresponding preposition, `prep1` and `prep2` respectively.

The noun-phrases can have `FIND` and search, `search-flags`, conditions defined. The token `FIND` means that the `OBJECT` must have the `flag-name` bit set. If there is only one `OBJECT` in the scope that meets the `FIND` condition the parser makes a GWIM ("Get what I mean"). For example if there is only one doore in the room with the `DOORBIT` set an `OPEN` assumes that you mean that door.

One special case of `FIND` is when there is no indirect `OBJECT` but the `SYNTAX` ends with a preposition. In these cases a special bit, `KLUDGEBIT` (or `ROOMBIT`), is used so that the player can type sentences like "turn machine on" (`<SYNTAX TURN OBJECT (FIND DEVICEBIT) ON OBJECT (FIND KLUDGEBIT) = V-TURN-ON>`).

The `search-flags` `HAVE`, `MANY` and `TAKE` define the following rules for the `OBJECT`:

`HAVE`	The `OBJECT` must be in the player's inventory (or inside open containers in the player's inventory). If the OBJECT is not in the inventory the parser fails and prints something like "You don't have the x,".
`MANY`	It is possible to use multiple `OBJECT`s with this `verb`.
`TAKE`	If the `OBJECT` is not in the player's inventory but takeble the parser attempts to take the `OBJECT`, an so called implicit take is performed, before continuing (the `OBJECT` is moved to the player's inventory and the parser prints something like "[Taken.]").

The `search-flags` `CARRIED`, `HELD`, `IN-ROOM` and `ON-GROUND` can be seen as hints to the parser where to first look for the `OBJECT`. These flags define the scope for the search. Note that these flags are only hints to the parser and if the `OBJECT` is not in the defined scope the parser continues the search in the other scopes before it fails. The default value for scope is that all flags

are set.

CARRIED	Search for the OBJECT inside open containers in the player's inventory.
HELD	Search for the OBJECT in the player's inventory at top-level (not inside other containers).
IN-ROOM	Search for the OBJECT inside containers on the ground.
ON-GROUND	Search for the OBJECT on the ground at the top-level.

Finally after the token = (equal-sign) there is one or two ROUTINE-names specified, action-routine-name and preaction-routine-name (optional). By convention these handlers are usually named V-verb and PRE-verb, respectively.

The preaction-routine-name is fired before the OBJECTs action-routine and the action-routine-name is fired after the OBJECTs action-routine. The preaction is usually used to check the prerequisites for the verb, for example that you have a weapon before attacking something so you don't have to check that in every attackable OBJECTs action-routine. The action-routine-name is usually used to handle response when the OBJECTs action-routine fails.

Each occurrence of an action-routine-name together with an optional preaction-routine-name must always have the same pattern (same action-routine-name can't exist with different preaction-routine-names).

It is possible to replace the search-flags with the GVAL NEW-SFLAGS. This is used with the new parser in Arthur, Shogun and Zork Zero where the search-flags ALL, ROOM, HELD, CARRIED, IN-ROOM, ON-GROUND, EVERYWHERE, MOBY and ADJACENT are defined.

Examples:

```
<SYNTAX QUIT = V-QUIT>
<SYNTAX CONTEMPLATE OBJECT = V-THINK-ABOUT>
<SYNTAX TAKE OBJECT (FIND TAKEBIT) (MANY ON-GROUND IN-ROOM)
     = V-TAKE>
<SYNTAX PUT OBJECT (MANY TAKE HELD CARRIED) IN OBJECT
     (FIND CONTBIT) = V-PUT-IN PRE-PUT-IN>
<SYNTAX WAKE OBJECT (FIND PERSONBIT) = V-WAKE>

<SYNTAX WAKE UP OBJECT (FIND PERSONBIT) = V-WAKE>
<SYNTAX WAKE OBJECT (FIND PERSONBIT) UP OBJECT
     (FIND KLUDGEBIT) = V-WAKE>
```

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`. See `ADD-TELL-TOKEN` for a description of `pattern` and `form`.

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>>
```

TIME

```
<TIME>

MDL built-in
```

ZILF ignores this and always returns `1`.

TOP

```
<TOP array>

MDL built-in
```

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 built-in
```

`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` doesn'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 `"OPT"` 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 built-in
```

evaluates to the type of `value`. See also `ALLTYPES`.

Examples:

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

TYPE?

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

MDL built-in
```

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>

MDL built-in
```

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 built-in
```

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>

MDL built-in
```

`UNPARSE` returns a `STRING` representation of `value`. Unlike `PNAME`, `UNPASE` prints an `ATOM`s trailers if required.

Examples:USE

```
<UNPARSE 123>            -->  "123"
<UNPARSE <+ 1 2>>        -->  "3"
<UNPARSE FOO>            -->  "FOO"
<UNPARSE <ATOM "FOO">>   -->  "FOO!-#FALSE ()"
<PNAME <ATOM "FOO">>     -->  "FOO"
```

USE

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

MDL package system
```

`USE` activates one or many `package-name`s and makes its content available in the current `OBLIST`-path. In practice `USE` copies the `OBLIST` `package-name` and adds it last to the local `OBLIST` (`<LVAL OBLIST>`). This means that all `ATOM`s on the external package `OBLIST` becomes available in current environment.

If the `package-name` is not available in the current environment, `USE` tries to load "`package-name.zil`" from the current path.

`USE` only works together with `PACKAGE` and if the definition of the `package-name` is missing from the environment or no file is found containing that definition is found, an error is raised.

See PACKAGE and USE-WHEN.

Example:

```
<USE "FOOFOO"> ;"Searches for file "foofoo.zil" which
                 contains the definition for
                 <PACKAGE "FOOFOO"> ..."
```

USE-WHEN

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

MDL package system
```

USE-WHEN is exactly like USE but only activates the package-name if the condition evaluates to TRUE.

See PACKAGE and USE.

Example:

```
<PACKAGE "FOO">
<SETG AAAA 1234>
<ENTRY AAAA>
<ENDPACKAGE>

<GASSIGNED? AAAA>                                  -->  #FALSE
<REMOVE AAAA> ;"Secure that ATOM not on any OBLIST"
<USE-WHEN <=? 1 2> "FOO">
<GASSIGNED? AAAA>                                  -->  #FALSE
<REMOVE AAAA> ;"Secure that ATOM not on any OBLIST"
<USE-WHEN <=? 1 1> "FOO">
,AAAA                                              -->  1234
```

VALID-TYPE?

```
<VALID-TYPE? atom>

MDL built-in
```

VALID-TYPE? returns the TYPE if the atom is a valid name of a TYPE (the atom name is in ALLTYPES), otherwise FALSE.

Examples:

```
<VALID-TYPE? VECTOR>     -->  VECTOR

<VALID-TYPE? FOO>        -->  #FALSE
<NEWTYPE FOO FIX>
<VALID-TYPE? FOO>        -->  FOO
```

VALUE

```
<VALUE atom [environment]>

MDL built-in
```

VALUE 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 built-in
```

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 an UVECTOR as a 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 ...>

ZIL parser library
```

VERB-SYNONYM creates one or more synonyms to the original verb.

ZILF treats VERB-SYNONYM as an alias to SYNONYM.

VERSION

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

ZIL library
```

This tells 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, Shogun 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`, holds 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]>

ZIL parser library
```

VOC inserts the string in the game vocabulary (dictionary). Normally there is no need to define the vocabulary with VOC, the vocabulary is automatically updated with words when you define ROOMs, OBJECTs, SYNTAX, etc.

What follows below is a description of the vocabulary when you use the standard parser library. The vocabulary description for the new parser (<SETG NEW-PARSER? T>) is in ADD-WORD.

The part-of-speech can be one of the following:

part-of-speech	Value	Description
<>	0	None
BUZZ	4	Buzz-word
PREP	8	Preposition
DIR	16	Direction
ADJ or ADJECTIVE	32	Adjective
VERB	64	Verb
NOUN or OBJECT	128	Noun

The vocabulary then occupies 6 or 9 bytes, depending on version, per entry distributed as follows.

Version 3

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
Word up to 6 Z-characters (5 bit)				PoS	Value	V2

Version 4-

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8

Word up to 9 Z-characters (5 bit)	PoS	V1	V2

PoS Byte 6 (or byte 4) contains the part-of-speech value (as above) plus if the word is defined as a first part-of-speech in the first 2 bytes.

```
0    None
1    Verb first
2    Adjective first
3    Direction first
```

V1 Byte 7 (or byte 5) contains the words value (id). Each part-of-speech can have 255 (65535 for NOUNs) unique words (synonyms have the same value as parent).

V2 Byte 8 is used for NOUNs (V1 & V2 gives 2 bytes, 1-65535 OBJECTs).

The different part-of-speech and first definitions have all global values defined as:

```
P1?OBJECT        0
P1?VERB          1
P1?ADJECTIVE     2
P1?DIRECTION     3
PS?BUZZ-WORD     4
PS?PREPOSITION   8
PS?DIRECTION    16
PS?ADJECTIVE    32
PS?VERB         64
PS?OBJECT      128
```

Example:

```
<VERSION XZIP>
<VOC "FALSE" <>>
<VOC "NOUN" NOUN>
<VOC "BUZZ" BUZZ>
<VOC "VERB" VERB>
<VOC "ADJECTIVE" ADJ>
<VOC "PREP" PREP>

<ROUTINE GO () <TEST-VOC> <INPUT 1>>

<ROUTINE TEST-VOC ("AUX" P)
     <SET P W?FALSE>
          <TELL "FALSE: Pos=" N <GETB .P 6>
               ", V1=" N <GETB .P 7>
               ", V2=" N <GETB .P 8> CR>
     <SET P W?NOUN>
          <TELL "NOUN: Pos=" N <GETB .P 6>
               ", V1=" N <GETB .P 7>
               ", V2=" N <GETB .P 8> CR>
     <SET P W?BUZZ>
          <TELL "BUZZ: Pos=" N <GETB .P 6>
               ", V1=" N <GETB .P 7>
               ", V2=" N <GETB .P 8> CR>
```

```
     <SET P W?VERB>
          <TELL "VERB: Pos=" N <GETB .P 6>
               ", V1=" N <GETB .P 7>
               ", V2=" N <GETB .P 8> CR>
     <SET P W?ADJECTIVE>
          <TELL "ADJECTIVE: Pos=" N <GETB .P 6>
               ", V1=" N <GETB .P 7>
               ", V2=" N <GETB .P 8> CR>
     <SET P W?PREP>
          <TELL "PREP: Pos=" N <GETB .P 6>
               ", V1=" N <GETB .P 7>
               ", V2=" N <GETB .P 8> CR>>

-->

FALSE: Pos=0, V1=0, V2=0
NOUN: Pos=128, V1=1, V2=0
BUZZ: Pos=4, V1=255, V2=0
VERB: Pos=65, V1=255, V2=0
ADJECTIVE: Pos=32, V1=0, V2=0
PREP: Pos=8, V1=255, V2=0
```

WARN-AS-ERROR?

```
<WARN-AS-ERROR? value>

ZILF compiler directive
```

`WARN-AS-ERROR?` set to `TRUE`, tells the compiler to convert compiler warnings to errors. The default value is `FALSE`.

Examples:

```
;"Compiles with warning [ZIL0204]"
<WARN-AS-ERROR? <>>
<GLOBAL X 5>
<ROUTINE GO () <TELL N .X>>

;"Don't compile with error [ZIL0204]"
<WARN-AS-ERROR? T>
<GLOBAL X 5>
<ROUTINE GO () <TELL N .X>>
```

XFLOAD

```
<XFLOAD filename>

ZIL library
```

ZILF ignores all but the first argument and treats `XFLOAD` as an alias to `INSERT-FILE`.

XORB

```
<XORB numbers ...>

MDL built-in
```

Bitwise exclusive "or".

Examples:

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

ZGET

```
<ZGET table index>

ZIL libarary
```

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 built-in `GET`.

Also see `GETB`, `PUTB`, `ZPUT` and `ZREST`.

Example:

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

ZIP-OPTIONS

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

ZIL library
```

`ZIP-OPTIONS` sets the corresponding bits in the header. This tells the Z-machine that runs the game, the interpreter, that the game intends to use these functions. The interpreter can, if it is unable to provide the requested functionality, clear the bits in return.

Option	Ver	Description
`BIG`	X	ZILF ignores this.
`COLOR`	5-	Sets bit 6 of byte 16 (Flags 2) in header (see Appendix B).
`DISPLAY`	5-	Sets bit 3 of byte 16 (Flags 2) in header (see Appendix B).
`MENY`	6-	Sets bit 8 of byte 16 (Flags 2) in header (see Appendix B).
`MOUSE`	5-	Sets bit 5 of byte 16 (Flags 2) in header (see Appendix B).
`SOUND`	5-	Sets bit 7 of byte 16 (Flags 2) in header (see Appendix B).
`UNDO`	5-	Sets bit 4 of byte 16 (Flags 2) in header (see Appendix B).

Example (From zillib/parser.zil in ZILF 0.9):

```
;"Use UNDO and COLOR if version is 5+"
<VERSION?
    (ZIP)
    (EZIP)
```

```
        (ELSE <ZIP-OPTIONS UNDO COLOR>)>
```

ZPACKAGE

```
<ZPACKAGE package-name>

ZIL library
```

ZPACKAGE is an alias to PACKAGE.

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 built-in 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 built-in 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]
```

ZSECTION

```
<ZSECTION package-name>
```

```
ZIL library
```

ZSECTION is an alias to DEFINITIONS.

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
```

ZSTR-OFF

```
<ZSTR-OFF>

ZIL library
```

ZILF ignores this and always returns FALSE.

ZSTR-ON

```
<ZSTR-ON>

ZIL library
```

ZILF ignores this and always returns FALSE.

ZZPACKAGE

```
<ZZPACKAGE package-name>

ZIL library
```

ZZPACKAGE is an alias to PACKAGE.

ZZSECTION

```
<ZZSECTION package-name>

ZIL library
```

ZZSECTION is an alias to DEFINITIONS.

Z-code built-ins (use inside ROUTINE)

Sources:

ZIP: Z-language Interpreter Program, Joel M. Berez, Marc S. Blank and P. David Lebling

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	**Inform syntax**
MUL	mul

Multiply `numbers`.

Example:

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

+, ADD

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

Zapf syntax	**Inform syntax**
ADD	add

All versions

Add `numbers`.

Example:

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

-, SUB

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

Zapf syntax	**Inform syntax**
SUB	sub

All versions

Subtract first `number` by subsequent `numbers`.

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

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

Example:

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

/, DIV

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

Zapf syntax	**Inform syntax**
DIV	div

All versions

Divide first number by subsequent numbers.

Example:

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

0?, ZERO?

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

Zapf syntax	**Inform syntax**
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"

Zapf syntax          Inform syntax
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"
```

Zapf syntax **Inform syntax**

```
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          Inform syntax
ASSIGNED?            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"
```

Zapf syntax	**Inform syntax**
BAND	and

All versions

Bitwise AND.

Examples:

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

BCOM

```
<BCOM value>
```

Zapf syntax	**Inform syntax**
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"
```

Zapf syntax	**Inform syntax**
BOR	or

All versions

Bitwise OR.

Examples:

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

BTST

```
<BTST value1 value2>
```

Zapf syntax	**Inform syntax**
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	**Inform syntax**
BUFOUT	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	**Inform syntax**
CATCH	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	**Inform syntax**
CHECKU	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	**Inform syntax**
CLEAR	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	**Inform syntax**
COLOR	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                                ;"Or T"
     <TELL "ELSE <...>" CR>
)>
```

`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          Inform syntax
COPYT                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          Inform syntax
CRLF                 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          Inform syntax
CURGET               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	**Inform syntax**
DCLEAR	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	**Inform syntax**
DEC	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>
```

Zapf syntax	**Inform syntax**
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-"
```

Zapf syntax	**Inform syntax**
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          Inform syntax
DISPLAY              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          Inform syntax
DLESS?               dec_chk

All versions
```

Predicate. Decrease variable (signed) `name` with 1and 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	**Inform syntax**
ERASE	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 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.

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	**Inform syntax**
FCLEAR	clear_attr

All versions

Removes `flag` from `object`.

Example:

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

FIRST?

```
<FIRST? object>
```

Zapf syntax	**Inform syntax**
FIRST?	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	**Inform syntax**
FONT	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 *The Z-Machine Standards Document*)
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	**Inform syntax**
FSET	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	**Inform syntax**
FSET?	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	**Inform syntax**
FSTACK	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	**Inform syntax**
GRTR?	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	**Inform syntax**
GET	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	Inform syntax
GETB	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	Inform syntax
GETP	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	Inform syntax
GETPT	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	**Inform syntax**
HLIGHT	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...) ...>
```

`IFFLAG` inside a `ROUTINE` have the same behaviour as `IFFLAG` outside.See `IFFLAG` (outside `ROUTINE`) for more information.

IGRTR?

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

Zapf syntax	**Inform syntax**
IGRTR?	inc_chk

```
All versions
```

Predicate. Increase variable (signed) name with 1and 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>
```

Zapf syntax	**Inform syntax**
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>
```

Zapf syntax	**Inform syntax**
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]>
```

Zapf syntax	**Inform syntax**
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 [rec-spec]>  ;"Version 5, 7-"
<INTBL? value table length>             ;"Version 4, 6"
```

Zapf syntax	**Inform syntax**
INTBL?	scan_table

Versions: 4-

INTBL? is a predicate that returns the address of value if value is in table of length, otherwise 0.

In version 5, 7 and 8 the rec-spec describes the field where bit 7 is set for words and clear for bytes, rest defines the length of the field.

Examples:

```
<T <INTBL? 3 <TABLE 1 2 3 4> 4>>   -->  T
<T <INTBL? 6 <TABLE 1 2 3 4> 4>>   -->  #FALSE

;"Search byte-table with record length 3 (ver 5, 7-)"
<T <INTBL? 8 <TABLE (BYTE) 2 0 1 4 0 1 8 0 1> 9 3>>    --> T
<T <INTBL? 1 <TABLE (BYTE) 2 0 1 4 0 1 8 0 1> 9 3>>    --> <>

;"Search word-table with record length 3 (ver 5, 7-)"
<T <INTBL? 8 <TABLE 2 0 1 4 0 1 8 0 1> 9 131>>         -->  T
```

```
<T <INTBL? 1 <TABLE 2 0 1 4 0 1 8 0 1> 9 131>>          --> <>
```

IRESTORE

```
<IRESTORE>
```

Zapf syntax	**Inform syntax**
IRESTORE	restore_undo

Versions: 5-

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

ISAVE

```
<ISAVE>
```

Zapf syntax	**Inform syntax**
ISAVE	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	**Inform syntax**
LESS?	Jl

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	**Inform syntax**
LEX	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	Inform syntax
LOC	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"
```

```
Zapf syntax          Inform syntax
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 ech 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	**Inform syntax**
MARGIN	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	**Inform syntax**
MENU	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	**Inform syntax**
MOD	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	**Inform syntax**
MOUSE-INFO	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	**Inform syntax**
MOUSE-LIMIT	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	**Inform syntax**
MOVE	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	Inform syntax
NEXT?	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>
```

Zapf syntax	Inform syntax
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	**Inform syntax**
ORIGINAL?	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	**Inform syntax**
PICINF	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	**Inform syntax**
PICSET	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	**Inform syntax**
POP	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	**Inform syntax**
PRINT	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	**Inform syntax**
PRINTB	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"
```

PRINTC

```
<PRINTC character>
```

Zapf syntax	**Inform syntax**
PRINTC	print_char

All versions

Print character.

Example:

```
<PRINTC 65>    -->  A
```

PRINTD

```
<PRINTD object>
```

Zapf syntax	**Inform syntax**
PRINTD	print_obj

All versions

Print description of object.

Example:

```
<GLOBAL MYOBJECT (DESC "sword">

<PRINTD ,MYOBJECT>  -->  "sword"
```

PRINTF

```
<PRINTF table>
```

Zapf syntax	**Inform syntax**
PRINTF	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	Inform syntax
PRINTI	print

All versions

Print `string`.

Example:

```
<PRINTI "Hello, sailor!">     -->  "Hello, sailor!"
```

PRINTN

```
<PRINTN number>
```

Zapf syntax	Inform syntax
PRINTN	print_num

All versions

Print number.

Example:

```
<PRINTN <+ 1 3>>    -->  4
<PRINTN -42>        -->  -42
```

PRINTR

```
<PRINTR string>
```

Zapf syntax	Inform syntax
PRINTR	print_ret

All versions

Print `string` and then `CRLF`.

Example:

```
<PRINTR "Hello, Sailor!">     -->  "Hello, sailor!\n"
```

PRINTT

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

Zapf syntax	Inform syntax
PRINTT	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	**Inform syntax**
PRINTU	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	**Inform syntax**
PTSIZE	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	**Inform syntax**
PUSH	push

All versions

Push `value` on game stack.

Example:

```
<PUSH 123>
```

PUT

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

Zapf syntax	**Inform syntax**
PUT	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	Inform syntax
PUTB	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>
```

Zapf syntax	Inform syntax
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>
```

Zapf syntax	Inform syntax
QUIT	quit

All versions

Halts game execution. No questions asked.

RANDOM

```
<RANDOM range>
```

Zapf syntax	Inform syntax

```
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          Inform syntax
READ                 aread / sread

All versions
```

Read text from the 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 `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 details about READ.

REMOVE

```
<REMOVE object>
```

Zapf syntax	**Inform syntax**
REMOVE	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          Inform syntax
RESTART              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-"

Zapf syntax          Inform syntax
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]>

Zapf syntax          Inform syntax
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         Inform syntax
RFALSE              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         Inform syntax
RSTACK              ret_popped

All versions
```

Pops value from game stack and returns that value.

Example:

```
<PUSH 42>
<RSTACK>       -->  Returns 42
```

RTRUE

```
<RTRUE>

Zapf syntax         Inform syntax
RTRUE               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	**Inform syntax**
SAVE	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

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

Zapf syntax	**Inform syntax**
SCREEN	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

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

Zapf syntax	**Inform syntax**
SCROLL	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

```
<SET name value>

Zapf syntax          Inform syntax
SET                  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          Inform syntax
SET                  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          Inform syntax
SOUND                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          Inform syntax
THROW                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 was "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>

Zapf syntax          Inform syntax
USL                  show_status

Versions: 3
```

Update status line. In other versions than 3 this command is ignored.

VALUE

```
<VALUE name/number>

Zapf syntax          Inform syntax
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>

Zapf syntax          Inform syntax
VERIFY               verify

All versions
```

Returns true if sum(`$0040:PLENTH` (byte 26-27 in header)) `MOD $10000` = `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>
```

Zapf syntax	**Inform syntax**
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 opertions are:

0: Set to flags
1: Set bits supplied (BOR)
2: Clear bits supplied
3: Reverse bits supplied

WINGET

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

Zapf syntax	**Inform syntax**
WINGET	get_wind_prop

Versions: 6-

Reads property on window-number.

WINPOS

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

Zapf syntax	Inform syntax
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	Inform syntax
WINPUT	put_wind_prop

Versions: 6-

Writes value to property window-number.

WINSIZE

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

Zapf syntax	Inform syntax
WINSIZE	window_size

Varsions: 6-

Changes size on window-number.

XPUSH

```
<XPUSH value stack>
```

Zapf syntax	Inform syntax
XPUSH	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	**Inform syntax**
ZWSTR	encode_text

Varsions: 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 specifikations 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 transcripting 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
MSLOCX	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

Name	**Type**	**Default value**	**Description**
CRLF-CHARACTER	GVAL	|	
DO-FUNNY-RETURNS?	GVAL	<> Versions 3-4 T Versions 5-	
FALSE-VALUE	CONSTANT	0	
FATAL-VALUE	CONSTANT	2	
IN-ZILCH	COMPILATION-FLAG	<>	
NEW-PARSER?	GVAL	Not defined	<SETG NEW-PARSER T> to use new parser
NEW-SFLAGS	GVAL		
PRESERVE-SPACES?	GVAL	<>	
REDEFINE	LVAL	<>	
SENTENCE-ENDS?	FILE-FLAG		