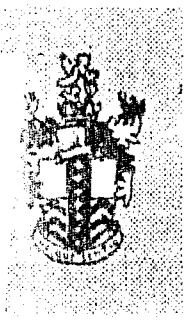
UNLIMITED



RSRE
MEMORANDUM No. 4356

ROYAL SIGNALS & RADAR ESTABLISHMENT

ZADOK USER GUIDE

Author: G P Randell

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
R S R E MALVERN.
WORCS.

R S R E MEMORANDUM No. 4356

UNLIMITED

CONDITIONS OF RELEASE

0067285 BR-113367

********** DRIC U

********** DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

********** DRIC N

DCAF CODE 090996

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 4356

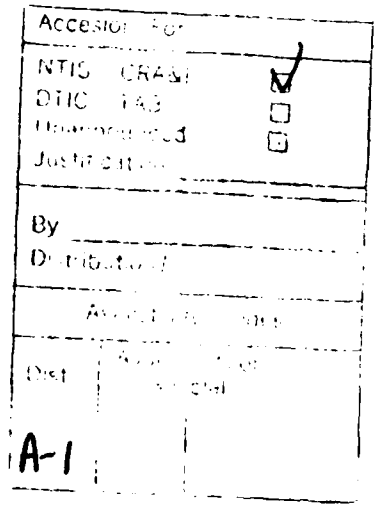
Title: ZADOK User Guide

Author: G P Randell

Date: January 1990

ABSTRACT

This is a guide for users of ZADOK, the RSRE Z syntax and typechecker. It also contains a brief introduction to using the Perq Flex system.



CONTENTS

1. Introduction

This document is a guide for users of Zadok, the Z syntax and typechecker developed by the SCEIP Unit at RSRE. It is assumed that users of this tool may not be familiar with the Perq computer operating with the Flex system, so an introduction to the use of Flex is given in Section 2.1. Some knowledge of Z is assumed.

This Guide also introduces the keyware software developed by the SMITE team at RSRE which is used with the Z tool. Other topics covered are how to use the Z editor; how to run the Z typechecker and correct errors it may find in your file of Z; and how to print your file using a PostScript printer. Section 6 discusses two issues relating to the Z language, namely the differences between the syntax used by the typechecker and that given in Spivey's reference manual [Spivey88], and the style of language adopted by the typechecker.

The final section describes some known problems with the Z tools, including the editor, and gives advice on how to deal with these. The annexes contain the Z symbols used by the editor, and what they mean, the Z library of basic mathematical constructs used by the typechecker and the Z syntax used by the tool.

2. Getting Started on the Perq

2.1 Using the Flex System

The Flex system provides on-line assistance for new users in the form of a tutorial and on-line documentation. When the Perq is switched on, there is an introductory display inviting you to log in. Instructions on how to login, and explanations of the elementary keyboard operations can be displayed by pressing the HELP key on the top left of the Perq keyboard. Basic terminology like puck and cursor is also explained. There is a user called *guest* which can be accessed by anyone from which a tutorial can be read and some experiments done (note the tutorial file can also be accessed from any user). To log in to any user name, for example, *your_name*, type

```
()your_name!
```

on a line by itself, and press ENTER on the keypad.

Other on-line documentation may be reached in several ways. There is an Edfile called doc which is shared and contains documentation on common modules and procedures. This is accessed by typing the name doc on a line by itself, obeying it by pressing Enter on the keypad, then pressing Examine (the centre button) on the puck. The top level of this file contains a cartouche (box) for each topic. Examining a cartouche will display documentation on that topic. Information may also be accessed for a named value by typing info.name on a line by itself, pressing ENTER on the keypad and then Examine on the puck.

Pressing the HELP key on the keyboard will display information depending on where you are. It is usually information on the structure you are pointing to with the cursor and the operations which may be performed on that structure. In these methods the editor is being used to examine the appropriate Edfile. Exiting from the documentation, as from any other file, is achieved by pressing Result (the right hand button) on the puck or Quit (CTRL-LINE FEED) on the keyboard.

The command interpreter language for the Flex system is called *Curt* [Currie82]. The most important function of Curt is to call procedures. These procedure calls are expressed in reverse Polish, that is, an expression which allows the parameter to be written before the procedure to be called is mentioned. For example, a procedure call which would be written f(x) in normal notation is expressed in Curt by:

```
x f!
```

The ! symbol says that the "thing" immediately before it (f) is a procedure and the previous "thing" (x) is its parameter. Pressing ENTER (also called *Obey*) evaluates the procedure and returns a value which will be represented on the screen by a box drawn around the procedure call. This box is called a *cartouche*. If the result of the procedure call is something that can be displayed, it may be examined by pressing Examine (the middle button) on the puck when the cursor is on the cartouche. If you press the Undo Cartouche key (5 on the key-pad) while the cursor is on the cartouche, one level of evaluation is undone. If you press the Mode key (1 on the keypad) while the cursor is on a cartouche the symbols in the box will be replaced by the mode of the value that the cartouche is standing for.

Values may be named, either temporarily or permanently. For example, there is an identifier known to the system, *a4_blank*, which is an *Edfile* (a Curt mode), and is a blank file of width equivalent to that of a piece of a4 paper, and capable of being extended indefinitely in length. Suppose you have created such a file, by typing a4_blank on a line by itself and evaluating it by pressing ENTER on the keypad. To type something into the file, press Examine on the puck with the cursor on the cartouche representing the file. This takes you in to the file, and automatically calls the editor. Having typed in some text, resulting from the file by pressing Result (the right hand button) on the puck will remember the changes you have made and the cartouche you see will represent the changed value. If you quit from the file, by pressing CTRL-LINE FEED, the changes will not be remembered and the cartouche

will represent the old value. To name the file, bill, say, obey the following command (that is, type the command on a line by itself and press ENTER on the keypad)

```
a4_blank = bill
```

This introduces the name "bill" as a temporary name for the file. Typing the name of a value on a line by itself and pressing ENTER will give you a cartouche standing for the same value. Names must start with a lower-case letter and contain lower-case letters, digits and the underline symbol. Temporary names will last for the current session, and if you re-use the name for something else the later value will override the earlier one. To name a value permanently, a double equals "==" should be used instead of the single equals sign for temporary names.

When you log in, you are calling the procedure your_name with a void parameter (written ()), which creates an environment for you to work in. After logging in for the first time, it is advisable to create a password for your user-name. This may be achieved by obeying the following command (that is, type the command on a line by itself and press ENTER on the keypad)

```
"fred" make_password!
```

which will create the password "fred" (using the procedure make_password with parameter fred (a vector of characters)). The next time you log in the Perq will ask you to type your password before logging you in. After typing the password, type a full stop to indicate the end of the password (not RETRN as you may expect). For more information see the information file info.make_password on the Perq. An *initial display* may also be created, which will be displayed each time you log in. For more information see info.initial_display on the Perq.

The environment of use is defined by the set of names accessible, and these are defined by the set of *dictionaries* available. A dictionary is a value of Curt mode Dictionary, and is a disc reference, that is, it is the only kind of value in filestore which can be updated. Each dictionary contains the association between names and values, the set of Modules belonging to the owner of the dictionary, and the history of alterations to the dictionary since the last garbage collection of the disc. One of the dictionaries accessible to you is your own user dictionary. This dictionary can be listed by obeying the following command

```
()show_dict!
```

An example of part of a dictionary is shown below.

DISPLAY	06/09/89 17 48 33	No Info
PASSWORD	11/09/89 15 01 11	No Info
amend	16/02/89 09 11 46	info.amend
z_spec	16/02/89 11 21 44	No Info

Associated with each value is the date and time it was last updated, together with an information file, if one exists.

Removing things from your own dictionary involves first listing the dictionary by using the *show_dict* command. To delete *z_spec*, say, place the cursor on the line starting with that name and press the Group key (6 on the keypad) until the cursor covers the entire line (not just the displayed characters and cartouches). The message at the top of the screen should be "(Box) Element (n) in Vertical". Then press Delete Element (PF4 on the keypad) to delete the line followed by Result on the puck to exit from show_dict. The name is not finally deleted until the command *tidy_dict* is applied to the result of the edit of the dictionary by typing tidy_dict! immediately to the right of the cartouche containing ()show_dict! and pressing Enter on the keypad. Old versions of values can be retrieved from the dictionary. For example, to recover old versions, of the file *z_spec*, say, obey the following command

```
"z_spec" old_versions!
```

which will display all versions of z_spec since the last garbage collection of the disk. More information on dictionaries may be found by examining the file Environments,dictionaries in doc.

Another dictionary accessible to you is the common dictionary which contains all named values that are shared by all users. To list this dictionary obey the following command

```
()show_common!
```

Most of the entries in this dictionary have an information file associated with them. These files may be examined in the usual way by placing the cursor on the cartouche and pressing Examine on the puck.

To keep the results of a session, explicit action must be taken. This means that values should be named permanently, that is, using two equals signs. This name/value association will then be automatically added to your dictionary. More information on keeping things may be found in the tutorial.

Pressing Result on the puck enough times will log you out back to the introductory display. If you accidentally log out without saving everything you need, you can recover if you immediately log in again by typing your user name followed by an exclamation mark to the right of the cartouche containing ()*your_name*!. This returns you to the display as it was when you accidentally logged out.

To create an Edfile in which you can write a Z specification, you can use an a4_blank. Alternatively, obeying the following command will give a file of a4 width, using Prop16 font (with a lead of 3), which is the font this Guide was written in, and which is required for printing a file of Z on a PostScript printer (see Section 7):

((5,3),(630,0))make_file!

This will give you an empty file, initially one line long but may be extended indefinitely in length, and pressing Examine on the puck will take you into the file. For information on the meaning of the parameters, see the file info.make_file on the Perq.

2.2 Installing the Z Tools

This section gives a brief guide for managers on installing the Z system from floppy disc.

Before installing the Z tools, two common users must be created in manager, called *z* and *keyware*. Installation uses the appropriate user name and consists of inputting a floppy disc file, reconstructing it and obeying the Curt lines (the ones with an exclamation mark) within them. For information on reconstructing, see info.make_reconstruct. In order to preserve the modules and allow for updating, keep the resulting files after undoing the Curt lines if necessary. The file name for the key compiler is *keys* and this may be installed independently of the Z system.

To install the Z system, it is necessary to install the Z editor first. As it declares the Z pictures, the Curt function *dec_picture_fns* must have been obeyed first. The file is called *zedit* and this must be copied in, reconstructed and installed before any other files are brought in. The syntax and type checker is split into two files called *ztc1* and *ztc2* because it occupies more than one floppy. The two files should be amalgamated before reconstructing and installing. The file *zmisc* simply contains a list of gblocks useful for Z, suitable for editing into a key compiler module for use with the *gblock_choose* function (see Section 3 for more information on the Keyware system).

3. Keyware

The key compiler system provides a very flexible way of assigning key functions to control keys. This also allows certain control keys to produce special frequently used characters such as "^" which are otherwise not available from the keyboard. Other keys may be set to bring up a table of special texts (a *gblock menu*) from which one text may be selected by placing the cursor on it and pressing Select (the left hand button) on the puck. For the Z editor, this feature is used to introduce schemas and other Z texts. Control keys may also be set to bring up a menu of useful functions, from which list one may be selected as before. An example of this is the main menu produced by pressing CTRL-a. This menu also gives access to a list of information files concerned with modules suitable for use with a control key.

An Edfile is first created containing the required keysetting modules. A typical example of such a file is given below.

```
std_keys

{ KEY a = main menu }
CHAR A = "∧"
KEY b = bold :Module                    { KEY B = Bottom of page }
KEY c = centre :Module                  KEY C = uncentre :Module
                          KEY D = subscript :Module
KEY e = enclose :Module                 KEY E = unenclose :Module
KEY f = forbid_entry :Module            KEY F = unforbid_m :Module
KEY g = group_char :Module    help_group :Module
KEY G = group_word :Module    help_group :Module
CHAR i = "∩"
KEY I = ( 6 , 3 ) "Italic"
KEY j = justify :Module                 KEY J = unjustify :Module
{ KEY k = list of keys }

CHAR m = "∈"     CHAR M = "∉"
CHAR n = "¬"     CHAR N = "ℕ"
CHAR o = "∨"     CHAR P = "ℙ"
KEY p = change_page :Module    help_change_page :Module

{ KEY P usually the laser printer }

CALL q,Q = qchoose :Module "Choose character from font"

CHAR t = "θ"

KEY u = underline :Module               KEY U = superscript :Module

CHAR v = "∪"    CHAR V = "∪"

APPLY y = gblock_choos :Module | z_gblocks | help_choose :Module

MENU "Handy Things" x
  = remove_dels :Module "Remove Dels"
  = overline :Module "Overline"
  = before_para :Module "Before Para"
  = unpara :Module "Unpara"
  = para :Module "Group to Para"
  = promote_file :Module "Edfile to Forbid"
  = show_size :Module "Show size"

FINISH
```

This file must be compiled using the key compiler, by obeying the following command

```
Edfile key_compiler!
```

The result of this is a *Compiledpair*, just like an implementation language compiler. The function *new* must then be applied as follows

```
Compiledpair new!
```

This will produce a module, called *std_keys* (the name at the top of the Edfile), which should then be used as the parameter of the function *keys* whenever the key settings need changing, by obeying the following command:

```
std_keys :Module keys!
```

Pressing ctrl-k after obeying this function will display which keys have been set. Those letters which have a blank after them have not been set. Using this example will produce the following:

CONTROL KEY FUNCTIONS

a ** main menu **	A ∧
b bold	B Bottom of Page
c centre	C uncentre
d	D subscript
e enclose	E unenclose
f forbi '_entry	F unforbid_m_
g group_char	G group_word
h	H
i ∩	I Italic
j justify	J unjustify
k ** This Help **	K
l	L
m ∈	M ∉
n ¬	N ℕ
o ∨	O
p change_page	P ℙ
q Choose character from font	Q Choose character from font
r	R
s	S
t θ	T Top of Page
u underline	U superscript
v ∪	V ∪
w	W
x Handy Things	X Repeat Handy Things
y Z_gblocks	Y
z	Z

A list of modules suitable for using in the key setting file may be found by examining the file Key procedu:·· inside Edfile utilities which may be found in doc.

4. The *Z* Editor

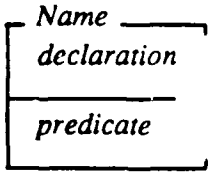
The Z editor is based on the Flex structure editing system called *pictures*. It uses a special Z picture to represent schemas and other Z boxes, theorems, *where*-phrases, and grouped Z text as required for indentation and global constraints. A Z picture can be incorporated into any editable structure. In preparing a specification an Edfile in whatever font and format is required for the documentation is used and the Z pictures inserted into it. Moving the cursor into a picture will automatically call the Z editor; normally this immediately re-calls the standard editor to edit the text of the signature or predicate as the case may be. These will always use the Z font, no matter what the outer font is. If the Z structure is to be changed (to name a schema for example) the Z editor must be invoked directly. This can be done by pressing the RETRN key. Alternatively, the Z editor may be invoked by the use of the Container key (0 on the keypad): press this repeatedly until the message at the top of the screen displays the Z structure in brackets, for example "(Signature) in Z box". There are three specific Z editing instructions provided, namely *insert blank*, *insert below*, and *help*.

insert blank (- on the keypad): if the structure being edited is either a signature in a Z box which allows a name or generic parameters, and these are not currently present, or a theorem without a hypothesis, an empty name or hypothesis is created and the cursor moved into it. If the structure is a predicate the editor beeps and does nothing, otherwise a blank line will be inserted above the entire Z phrase.

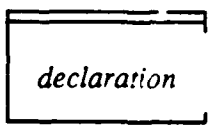
insert below (. on the keypad): if the structure being edited is a signature in a Z box which does not have a predicate, a blank predicate line is created and the cursor moved into it. Otherwise the editor beeps or inserts a line below the entire Z phrase as appropriate.

help (on the keyboard): displays a help file.

The Z pictures are as follows:



Schema box



Unique Definition box

declaration ——————— *predicate*	Vertical rule
⊢ *conclusion*	Theorem
predicate *where* *declaration*	*where*-phrase
[*set1*,*set2*]	Grouped Z text (indicated by a box round the Z text, but only when it is being edited)

Anything not in one of these pictures will be treated as surrounding text and ignored by the syntax and typechecker.

There is one special key, namely the LINE FEED key, which has a special meaning. It corresponds to a "hard new line", or vertical list separator in Z. The effect of pressing this key is to add a small invisible separator to the bottom of the current signature, predicate or Z phrase and to start a new line of characters. This separator, called a *white bar*, can be detected by placing the cursor between two lines separated in this way. The message "(White Bar) Element(n) in Vertical" will appear at the top of the screen. Note that the white bar is always added to the bottom of the Z, so must be deleted and inserted if it is required any where else. Only one hard new line must be used between lines of Z, else the typechecker will report an error.

5. Using the Typechecker

Having edited a Z specification, the next stage is to carry out syntax and type checking. This section deals with the practical aspects of how to call the typechecker, and the sort of results to expect. It explains how the tool displays error messages, the sort of messages to expect and how to correct errors.

5.1 Running the Typechecker

Calling the typechecker is a straightforward operation, and simply involves applying the function called *z* to an Edfile which contains Z, by obeying the following command:

Edfile z!

Errors in the Z specification will cause the typechecker to report faults using the Z editor as described in the following section. An error-free specification will result in either the original file or a *Compiledpair* in the case where the specification ends with a *keeps* statement. In the latter case the *Compiledpair* may be converted into a module

by using the *new* function or used to amend an existing module. The use of modules is further explained in section 5.4.

5.2 Error Messages

There are two types of error messages, those relating to syntax errors, and those relating to type errors. Syntax errors all have the same format, and are best illustrated by example. Consider the following part specification of a library system. This library uses two given sets, *BOOK* and *PERSON* to represent all the books in the world, and all the people in the world respectively. Suppose we had typed

$[BOOK; PERSON]$

with a semi-colon rather than a comma separating the two given sets. The Z tool will place the cursor on the offending semi-colon, and display the message

"**Syntax error, found** semi-colon, **expecting** comma **or** c square brkt"

at the top of the screen. In this message, "c square brkt" is an abbreviation of "close square bracket" ("]"). The words in bold give the format of all syntax error messages. If there are several possible symbols expected in place of the one found, then the error message will be longer than it is possible to display at the top of the screen. In this case, the message will appear in a box one line of text high, and as long as necessary, which is likely to disappear off the right of the screen. In this case, the complete message may be read by scrolling along the box using either the right arrow key or by holding the *select* button on the puck down and moving the arrow on screen to the right of the message box.

There is a variety of possible error messages resulting from a typechecking error. As an example, suppose the library specification continues with the following schema representing the state of the library:

Library

$$borrowers : \mathbb{P}\, PEOPLE$$
$$books : \mathbb{P}\, BOOK$$
$$books\_on\_shelves . \mathbb{P}\, BOOK$$
$$on\_loan : BOOK \nrightarrow PEOPLE$$

$$dom\ on\_loan \cap books\_on\_shelves = \emptyset$$
$$dom\ on\_loan \cup books\_on\_shelves = books$$
$$rng\ on\_loan \subseteq borrowers$$

The typechecker would find 2 errors in this schema. The first would be in the line

$borrowers : \mathbb{P}\ PEOPLE$

the cursor would be placed on the "*E* " of "*PEOPLE*" and the message "Identifier PEOPLE undeclared" would appear at the top of the screen. The second error is found in the line

$on\_loan : BOOK \pfun PEOPLE$

and the cursor placed on the "*E*" of "*PEOPLE*". The message at the top of the screen would this time be "Incorrect type for instantiation". The source of both of these messages is the fact that *PEOPLE* has not been declared anywhere. The second message reflects the fact that the typechecker is trying to instantiate $\pfun$, as this has been defined with generic parameters in the Z library, and *PEOPLE* is not a type. Note that the typechecker usually places the cursor on or after the error. Now suppose that the library specification is further developed by adding the following schema to describe the operation of borrowing a book:

Borrow

$p? : PEOPLE$

$b? : BOOK$

$Library; Library'$

$p? \in borrowers$

$b? \in books\_on\_shelves$

$books\_on\_shelves' = books\_on\_shelves \setminus \{b?\}$

$borrowers' = borrowers$

$books' = books$

$on\_loan' = on\_loan \cup \{b? \mapsto p?\}$

The typechecker will again find fault with *PEOPLE*, this time for two reasons: one, that it has not been declared, and two, that it is not a type, which is required in this position. This leads to two messages, which, as there is not enough room at the top of the screen for two lines, appear in a box in the main part of the file, immediately above the error. These messages are "Identifier PEOPLE undeclared" and "The term given is not a type".

In the case where the typechecker gives a message which includes the name of the offending identifier, then this will not include any decoration, such as ′,?, or !. This means that it could be referring to the identifier itself, or a decorated form of the identifier. It is important to note that the error messages do not themselves use the Z font, so if the identifier causing an error uses a symbol not available in the message font, a blank will appear in the message instead of the actual identifier.

5.3 Correcting Errors

Errors may be corrected one at a time, using the *tab* key to progress through the file. Care should be taken as some errors may be consequential, that is, by correcting one error several later errors may be corrected automatically. This often occurs with errors in defining a schema, as wherever that schema is used later on, the typechecker will not recognise it which will result in several consequential error messages. Another example of this is in the library specification of section 5.2 above. If the first (syntax) error prompted a change to [*BOOK, PEOPLE*], then none of the other errors would require any action.

Correcting errors is simply a matter of editing the file. However, the typechecker will mark a line containing an error which has the effect that if correcting the error requires any symbol from the Z gblock menu to be chosen, then it will not be inserted immediately before the cursor but will instead be placed at the beginning of the line. This is an unfortunate feature of the Flex editor. To overcome it, there is a key procedure called *unset_mark*, which can be used with CTRL-a or to define a control key, which will remove the mark from the line and allow it to be edited normally.

In the case where the error message appears in a box in the main part of the text, the *result* button on the puck must be clicked to free the cursor and put it back in the main body of the file before attempting to correct the error, and also before using the *tab* key to go to the next error. When all errors have been found, the message "Nothing found" will appear at the top of the screen. The file should then be typechecked again, and the process repeated until the specification is correct. Please note that just because a specification has been successfully typechecked it does not mean that the specification is consistent or that it specifies what was required.

5.4 Modules

Modules of Z are a means of structuring specifications. They allow large specifications to be built up and typechecked gradually, and also provide for some reusability of Z specifications. A module is formed by adding a *keeps* statement to the end of a specification, which has the form

module_name keeps id1, id2, id3

where the identifiers that appear in the *keeps* list are those which will be visible outside the module. This statement must be inside a grouped Z text picture.

For example, suppose we have corrected the library specification, and wish to make it into a module so that it may be used as part of a larger specification. The identifiers we wish to keep are *BOOK*, *PEOPLE*, *Borrow*, and *Library*, and we will call the module *library*. The specification is as follows:

$[BOOK, PEOPLE]$

Library

$$borrowers : \mathbb{P}\, PEOPLE$$
$$books : \mathbb{P}\, BOOK$$
$$books\_on\_shelves : \mathbb{P}\, BOOK$$
$$on\_loan : BOOK \nrightarrow PEOPLE$$

$$dom\ on\_loan \cap books\_on\_shelves = \varnothing$$
$$dom\ on\_loan \cup books\_on\_shelves = books$$
$$rng\ on\_loan \subseteq borrowers$$

Borrow

$$p? : PEOPLE$$
$$b? : BOOK$$
$$Library; Library'$$

$$p? \in borrowers$$
$$b? \in books\_on\_shelves$$
$$books\_on\_shelves' = books\_on\_shelves \setminus \{b?\}$$
$$borrowers' = borrowers$$
$$books' = books$$
$$on\_loan' = on\_loan \cup \{b? \mapsto p?\}$$

library keeps BOOK, PEOPLE, Borrow, Library

The result of typechecking this specification will be a *Compiledpair*, and this may be seen by pressing the mode key (1 on the keypad). The function *new* should then be applied by obeying the following command:

Compiledpair new!

A module will result, and pressing the mode key again will produce:

library :Module

Modules of Z may be incorporated into other files of Z by listing them at the top of the new file. Any number of modules may be listed, and an empty Z phrase must be put at the bottom of the list (below the last module). Any identifier appearing in the *keeps* list of one of the modules may then be used in the new file. It is important to note

that all identifiers required must be in the list, including all branches of datatype definitions. Keeping the name of a datatype will not automatically keep all its components.

A *Compiledpair* may, instead of having the function *new* applied to it to create a new module, be used to amend an existing module. The form of this command is

| Compiledpair | library :Module | amend!!

For further information on the *amend* command, see the information file info.amend on the Perq (which also gives information on modules and other functions which may be applied to them). An alternative to *amend* is the function *change_spec*, which must be used in the case where the *keeps* list has been altered or the type of any identifier changed. The format of the command line is the same as for *amend*, however all occurrences of the module changed will be flagged as "untidy". This may be checked by pressing the Mode key (1 on the keypad) with the cursor on an occurrence if in doubt. Every (untidy) occurrence must be tidied using the procedure *tidy_module*. See info.tidy_module for more information. Any specification which uses a module which has been changed using *amend* or *change_spec* will need to be re-typechecked.

6. Language Issues

The Z syntax used by the Z tool is based on that of King et al [King87], and is different in some respects from that published in [Spivey89]. This section describes some of the main differences between the syntaxes, and also discusses some features of the language as used by the Z tool. The type checking and scope rules of Z as used by the syntax and typechecking tool have been formally specified in Z, and the interested reader is referred to [Sennett87] for details.

6.1 Differences with Reference Manual Z

There are 3 major differences between the syntax used by the tool (see Annex C) and that of the Reference Manual, namely the use of theorems, *where* phrases and modules by the typechecker which are not part of the Reference Manual Z.

Theorems take the form

hypothesis
⊢
conclusion

and asserts that the *conclusion* may be derived from the *hypothesis*. The symbol ⊢ is called a turnstile. This construct is often used for stating proof opportunities.

The hypothesis may be empty, which is equivalent to *true*, or a list containing any number of predicates (which include schema references), declarations, and

declarations bar predicates. The items in the list are separated by either semi-colons or hard new lines. The conclusion of a theorem is a predicate.

where-phrases are a type of existential quantification, and take the form

Predicate
where
Declaration

The *predicate* part of a *where*-phrase is a list of predicates, while the *declaration* part may have one of the following forms: a list of syntactic definitions separated by semi-colons or hard new lines; one declaration bar predicate; or a vertical rule. As an example, consider the following *where*-phrase:

predicate
where

$$x : X \mid p(x)$$

This is equivalent to

$$\exists x : X \mid p(x) \bullet predicate$$

where-phrases are particularly useful in the predicate part of schemas, however care must be taken with the scoping of variables introduced in the the declaration part of a *where*-phrase, as such variables only have scope within the predicate part of the *where*-phrase in which they were introduced.

Modules have been introduced as a means of structuring Z specifications, much like using separately compiled pieces of code to build up a large computer program. See section 5.4 for details on how to create and use modules.

A further difference with Reference Manual Z is the omission of bags from the syntax used by the Z tool. These, together with operations on them, can be defined by the user if required. The most convenient way is by creating a module which can then be incorporated into any specification which requires it. It is also worth noting that the horizontal form of schemas is also not allowed by the typechecker.

6.2 Common Causes of Errors

The purpose of this section is to list some of techniques of writing Z adopted by the typechecker which should help to diagnose some of the more common errors. The incorrect matching of brackets often causes problems, with the error message "Incorrect type for function application or relation" being the most likely result. Another common problem is caused by the incorrect use of the hard new line - it is easy to leave one at the bottom of a Z phrase, or to use two, one under the other. In either case a syntax error will result, however the cursor will be placed *before* the extra hard new line by the typechecker not after the error as is usually the case. Other things to watch out for are:

1. Cross-products. When using cross products in signatures, they must always be bracketted, for example

$$[X, Y]$$
$$fst : (X \times Y) \rightarrow X$$

2. Infix functions and relations. When declaring the type of an infix function, brackets must be used round the function name and placeholders. However, brackets must not be used when declaring an infix relation, for example

$$[X, Y]$$
$$(\_ \mapsto \_) : (X \times Y) \rightarrow (X \times Y)$$

is a function, and

$$[X]$$
$$\_ \subseteq \_ : \mathbb{P} X \leftrightarrow \mathbb{P} X$$

is a relation.

3. Inverse. There is a special symbol for functional inverse, namely $^{-1}$, so superscripting is not required for this. (Superscripts may be used for iteration and user-defined operators.)

4. Renaming and instantiation. Lists for renaming and instantiation must be subscripted, for example

Schema1
$$n : \mathbb{N}$$

$Schema2 \triangleq Schema1_{[m/n]}$

5. Symbol overloading. Symbols may not be overloaded, that is, used more than once with different meanings.

7. Printing a File of Z Using the PostScript Printer

In order to print a file containing Z on the Apple LaserWriter II, the outermost font of the file must be Prop16 (that is, font (5,lead)). If the file was created using a different font, then it must be changed, and obeying the following command (which uses a lead of 3) will achieve the desired result:

Edfile , (5,3)) change_font!

For further information on this command, use the information file info.change_font on the Perq.

In order to print a file, it must first be put inside another Edfile, an a4_blank for example. Alternatively, there is a prepared file, called an *Apple_blank* which contains a Prop16 file inside another Prop16 file. The inner file is then used for writing Z. Obeying the following command will convert the required file into the PostScript format, and then send it to the Apple LaserWriter to be printed:

a4_blank , 12) post_script! apple_send!

The number, 12 in this example, determines the size of the printed characters in points. The smaller the number the smaller the print. This Guide was printed using 12.

More than one file can be printed at a time if required. This is particularly useful for printing chapters or sections of a document each prepared in its own file, as each file is started on a new page, and the pages numbered consecutively. It is achieved by putting all the files in a vertical list in the required order inside the outer Edfile, and obeying the same command as before.

It should be noted that the program used for printing Z as described is under development - *caveat emptor*.

8. Known Problems

This section lists the known problems with the Z editor and typechecker, and offers advice on how they may be overcome. If you discover any problems not listed here, please inform the author.

1. Modules inside a Z phrase. The problem here involves the editor, and occurs when a single module is placed inside a Z phrase. If the cursor is on the module, and the

right arrow used to move to the right, then the editor enters an infinite loop. The visible sign of this is the cursor disappearing, and a flickering of the module cartouche. Pressing CTRL REJ/DEL (break in) stops the looping, unfortunately all editing done since the last save is lost.

2. Overwriting the schema box. The problem here involves the display, and may occur when inserting elements into schemas where the schema contains a where phrase. The remembered element may be inserted over the bottom line of the schema box, that is, the bottom line is overwritten. Using insert remembered horizontal rather than vertical sometimes solves the problem, otherwise inserting a line at a time may help. If all else fails, the Z will have to be retyped in the required place.

3. Deleting structures. The problem which arises in this case occurs when a whole section of a Z picture, say the declaration part of a schema box, is deleted. If an attempt is made to put the cursor in the empty space, the cursor will disappear and the message "(White Bar) Signature in Z Box" will appear at the top of the screen. Typing something then pressing right arrow (top right of the keyboard) will remove the white bar and restore the line.

4. Using the typechecker in conjunction with the sketches software. If a sketch appears in a file containing Z, then when typechecked the Z tool will stop on the sketch. The TAB key must be pressed to move on. This applies when using forbids, too.

5. Inverse. If a functional inverse symbol is placed next to a left image bracket. that is, $^{-1}[$, without an intervening space, the typechecker will report an error with the message "identifier undeclared".

References

[Currie82]
"Curt: The Command Interpreter Language for Flex", I F Currie and J M Foster. RSRE Memorandum 3522 (September 1982)

[King87]
"Z: Grammar and Concrete and Abstract Syntaxes", S King, I H Sorensen and J Woodcock. Version 1.1, July 28th 1987, Programming Research Group, University of Oxford (July 1987)

[Sennett87]
"Review of Type Checking and Scope Rules of the Specification Language Z", C T Sennett. RSRE Report Number 87017 (November 1987)

[Spivey89]
"The Z Notation - A Reference Manual", J M Spivey. Prentice-Hall International Series in Computer Science (1989)

Annex A - The Z Symbols

This annex gives a list of symbols for Z, together with their meaning.

Symbol	Meaning
$\triangleq$	Schema definition
$\langle\!\langle$	Left angle bracket (for datatype definitions)
$\rangle\!\rangle$	Right angle bracket (for datatype definitions and for piping)
$\neg$	Not
$\wedge$	And
$\vee$	Or
$\Rightarrow$	Implication
$\Leftrightarrow$	Equivalence (if and only if)
$\forall$	For all
$\exists$	There exists
$\exists_1$	There exists (unique)
$\bullet$	Then
$\in$	Set membership
$\times$	Cartesian product
$\mathbb{P}$	Power set
$\fatsemi$	Forward composition
$\oplus$	Overriding
$\neq$	Not equals
$\cap$	Intersection
$\cup$	Union
$\subseteq$	Subset
$\subset$	Proper subset
$\notin$	Not member
$\bigcap$	Distributed intersection
$\bigcup$	Distributed union
$\mathbb{F}$	Finite set
$\emptyset$	Empty set
$\leftrightarrow$	Relation
$\lhd$	Domain restriction
$\not\lhd$	Domain subtraction
$\rhd$	Range restriction
$\not\rhd$	Range subtraction
$(\!\mid$	Left image bracket
$\mid\!)$	Right image bracket
$^{-1}$	Inverse
$\mapsto$	Maplet
$\mathbb{N}$	Natural numbers
$\geq$	Greater than or equals to
$\leq$	Less than or equals to
$^\frown$	Concatenation
$\upharpoonright$	Range restriction (for sequences)

$\langle$	Left sequence bracket
$\rangle$	Right sequence bracket
$\vdash$	Turnstile
$\nrightarrow$	Partial function
$\rightarrow$	Total function
$\nrightarrow\!\!\!\!\!\!+$	Finite function
$\rightarrowtail\!\!\!\!\!\!\nrightarrow$	Partial injection
$\rightarrowtail$	Total injection
$\rightarrowtail\!\!\!\!\!\!\nrightarrow\!\!\!\!\!\!+$	Finite injection
$\nrightarrow\!\!\!\!\!\!\twoheadrightarrow$	Partial surjection
$\rightarrow\!\!\!\!\!\!\twoheadrightarrow$	Total surjection
$\rightarrowtail\!\!\!\!\!\!\twoheadrightarrow$	Bijection

Annex B - The Z Library

SET OPERATIONS

RELATION SYMBOL

$$[X, Y]$$

$$X \leftrightarrow Y == \mathbb{P}(X \times Y)$$

$$X \rightarrow Y == \{f : X \leftrightarrow Y \mid \forall x : X \bullet \exists_1 y : Y \bullet (x, y) \in f\}$$

NEGATIONS

$$[X]$$

$$\_ \neq \_ : X \leftrightarrow X$$

$$\_ \notin \_ : X \leftrightarrow \mathbb{P} X$$

$$\forall x, y : X \bullet x \neq y \Leftrightarrow \neg(x = y)$$

$$\forall x : X; S : \mathbb{P} X \bullet x \notin S \Leftrightarrow \neg(x \in S)$$

$$[X]$$

$$\vdash$$

$$\forall x, y : X \bullet x \neq y \Rightarrow y \neq x$$

NULL SET

$$[X]$$

$$\varnothing : \mathbb{P} X$$

$$\varnothing = \{x : X \mid \neg(x = x)\}$$

$$[X]\ x : X$$

$$\vdash$$

$$x \notin \varnothing$$

SUBSET RELATIONS

$$[X]$$

$$\_ \subseteq \_, \_ \subset \_ : \mathbb{P} X \leftrightarrow \mathbb{P} X$$

$$\forall S, T : \mathbb{P} X$$

$$\bullet\ (S \subseteq T \Leftrightarrow (\forall x : X \bullet x \in S \Rightarrow x \in T))$$

$$\wedge\ (S \subset T \Leftrightarrow S \subseteq T \wedge S \neq T)$$

UNIONS AND INTERSECTIONS

$$[X]$$

$$(\_ \cup \_), (\_ \cap \_), (\_ \setminus \_) : (\mathbb{P} X \times \mathbb{P} X) \rightarrow \mathbb{P} X$$

$$\forall S, T : \mathbb{P} X$$

$$\bullet\ S \cup T = \{x : X \mid x \in S \vee x \in T\}$$

$$S \cap T = \{x : X \mid x \in S \wedge x \in T\}$$

$$S \setminus T = \{x : X \mid x \in S \wedge x \notin T\}$$

NON-EMPTY SUBSETS

$$[X]$$

$$\mathbb{P}_1 X == \mathbb{P} X \setminus \{\varnothing\}$$

GENERALIZED UNION AND INTERSECTION PROJECTION FUNCTIONS FOR PAIRS

$$[X]$$

$$\bigcup, \bigcap : \mathbb{P}\,\mathbb{P}\,X \to \mathbb{P}\,X$$

$$\forall A : \mathbb{P}\,\mathbb{P}\,X$$
$$\bullet \bigcup A = \{x : X \mid (\exists S : A \bullet x \in S)\}$$
$$\bigcap A = \{x : X \mid (\forall S : A \bullet x \in S)\}$$

$$[X, Y]$$

$$fst : (X \times Y) \to X$$
$$snd : (X \times Y) \to Y$$

$$\forall x : X; y : Y \bullet fst(x, y) = x \land snd(x, y) = y$$

RELATIONS

MAPLET

$$[X, Y]$$

$$(\_ \mapsto \_) : (X \times Y) \to (X \times Y)$$

$$\forall x : X; y : Y \bullet x \mapsto y = (x, y)$$

DOMAIN AND RANGE

$$[X, Y]$$

$$dom\_ : (X \leftrightarrow Y) \to \mathbb{P}\,X$$
$$rng\_ : (X \leftrightarrow Y) \to \mathbb{P}\,Y$$
$$ran\_ : (X \leftrightarrow Y) \to \mathbb{P}\,Y$$

$$\forall R : X \leftrightarrow Y$$
$$\bullet\ dom\,R = \{x : X; y : Y \mid R(x, y) \bullet x\}$$
$$\land\ rng\,R = \{x : X; y : Y \mid R(x, y) \bullet y\}$$
$$\land\ ran\,R = \{x : X; y : Y \mid R(x, y) \bullet y\}$$

IDENTITY RELATION

$$[X]$$

$$id\,X == \{x : X \bullet x \mapsto x\}$$

RELATIONAL COMPOSITION

$$[X, Y, Z]$$

$$(\_ \fatsemi \_) : ((X \leftrightarrow Y) \times (Y \leftrightarrow Z)) \to (X \leftrightarrow Z)$$

$$\forall R : X \leftrightarrow Y; S : Y \leftrightarrow Z$$
$$\bullet\ R \fatsemi S = \{x : X; y : Y; z : Z \mid R(x, y) \land S(y, z) \bullet x \mapsto z\}$$

RESTRICTION OPERATORS

$[X, Y]$

$(\_ \lhd \_) : (\mathbb{P} X \times (X \leftrightarrow Y)) \rightarrow (X \leftrightarrow Y)$

$(\_ \rhd \_) : ((X \leftrightarrow Y) \times \mathbb{P} Y) \rightarrow (X \leftrightarrow Y)$

$\forall S : \mathbb{P} X; R : X \leftrightarrow Y$

$\bullet S \lhd R = \{x : X; y : Y \mid x \in S \land R(x, y) \bullet x \mapsto y\}$

$\forall R : X \leftrightarrow Y; T : \mathbb{P} Y$

$\bullet R \rhd T = \{x : X; y : Y \mid R(x, y) \land y \in T \bullet x \mapsto y\}$

$[X, Y]$

$(\_ \lhd \_) : (\mathbb{P} X \times (X \leftrightarrow Y)) \rightarrow (X \leftrightarrow Y)$

$(\_ \rhd \_) : ((X \leftrightarrow Y) \times \mathbb{P} Y) \rightarrow (X \leftrightarrow Y)$

$\forall S : \mathbb{P} X; R : X \leftrightarrow Y$

$\bullet S \lhd R = \{x : X; y : Y \mid x \notin S \land R(x, y) \bullet x \mapsto y\}$

$\forall R : X \leftrightarrow Y; T : \mathbb{P} Y$

$\bullet R \rhd T = \{x : X; y : Y \mid R(x, y) \land y \notin T \bullet x \mapsto y\}$

RELATIONAL INVERSION

$[X, Y]$

$\_^{-1} : (X \leftrightarrow Y) \rightarrow (Y \leftrightarrow X)$

$\forall R : X \leftrightarrow Y \bullet R^{-1} = \{x : X; y : Y \mid R(x, y) \bullet y \mapsto x\}$

RELATIONAL IMAGE

$[X, Y]$

$\_(\!|\_|\!) : ((X \leftrightarrow Y) \times \mathbb{P} X) \rightarrow \mathbb{P} Y$

$\forall R : X \leftrightarrow Y; S : \mathbb{P} X$

$\bullet R(\!|S|\!) = \{x : X; y : Y \mid x \in S \land R(x, y) \bullet y\}$

TRANSITIVE CLOSURE

$[T]$

$\_^+, \_^* : (T \leftrightarrow T) \rightarrow (T \leftrightarrow T)$

$\forall R : T \leftrightarrow T$

$\bullet R^+ = \bigcap \{Q : T \leftrightarrow T \mid R \subseteq Q \land Q \fatsemi Q \subseteq Q\}$

$\land R^* = R^+ \cup id\, T$

FUNCTIONS

PARTIAL FUNCTIONS

$$[X, Y]$$

$$X \pfun Y == \{f : X \rel Y \mid \forall x : X;\ y_1, y_2 : Y \bullet f(x, y_1) \land f(x, y_2) \Rightarrow y_1 = y_2\}$$

INJECTIONS

$$[X, Y]$$

$$X \pinj Y == \{f : X \pfun Y \mid \forall x_1, x_2 : dom\, f \bullet f\, x_1 = f\, x_2 \Rightarrow x_1 = x_2\}$$

$$X \inj Y == (X \pinj Y) \cap (X \fun Y)$$

SURJECTIONS AND BIJECTIONS

$$[X, Y]$$

$$X \psurj Y == \{f : X \pfun Y \mid ran\, f = Y\}$$

$$X \surj Y == (X \psurj Y) \cap (X \fun Y)$$

$$X \bij Y == (X \surj Y) \cap (X \inj Y)$$

FUNCTIONAL OVERRIDE

$$[X, Y]$$

$$(\_ \oplus \_) : ((X \pfun Y) \times (X \pfun Y)) \fun (X \pfun Y)$$

$$\forall f, g : X \pfun Y \bullet f \oplus g = ((dom\, g) \ndres f) \cup g$$

NUMBERS AND ITERATION

NUMBERS

$$\mathbb{N} : \mathbb{P}\, \mathbb{Z}$$

$$(\_+\_), (\_-\_), (\_*\_) : (\mathbb{Z} \times \mathbb{Z}) \fun \mathbb{Z}$$

$$(\_div\_), (\_mod\_) : (\mathbb{Z} \times \mathbb{Z}) \fun \mathbb{Z}$$

$$\_<\_, \_\leq\_, \_\geq\_, \_>\_ : \mathbb{Z} \rel \mathbb{Z}$$

$$\sim\_ : \mathbb{Z} \fun \mathbb{Z}$$

$$\mathbb{N} = \{n : \mathbb{Z} \mid n \geq 0\}$$

$$\mathbb{N}_1 == \mathbb{N} \setminus \{0\}$$

$$succ : \mathbb{N} \fun \mathbb{N}$$

$$\forall n : \mathbb{N} \bullet succ\, n = n + 1$$

ITERATION

$$[X]$$

$$\_^{\sim} : ((X \rel X) \times \mathbb{Z}) \fun (X \rel X)$$

$$\forall R : X \rel X$$

$$\bullet R^0 = id\, X$$

$$\land (\forall k : \mathbb{N} \bullet R^{k+1} = R \fatsemi R^k)$$

$$\land (\forall k : \mathbb{Z} \bullet R^{-k} = (R^{-1})^k)$$

NUMBER RANGE

$$(\_..\_) : (\mathbb{Z} \times \mathbb{Z}) \rightarrow \mathbb{P}\,\mathbb{Z}$$

$$\forall\, a, b : \mathbb{Z} \bullet a..b = \{k : \mathbb{Z} \mid a \leq k \leq b\}$$

FINITE SETS AND CARDINALITY

$$[X]$$

$$\mathbb{F}\,X == \{S : \mathbb{P}\,X \mid \exists\, n : \mathbb{N} \bullet \exists f : 1..n \rightarrow S \bullet rng\, f = S\}$$
$$\mathbb{F}_1 X == \mathbb{F}\,X \cap \mathbb{P}_1\, X$$

$$[X]$$

$$\#\_ : \mathbb{F}\,X \rightarrow \mathbb{N}$$

$$\forall\, S : \mathbb{F}\,X$$
$$\bullet\ \#S = \mu\, n : \mathbb{N} \mid \exists f : 1..n \rightarrowtail S \bullet rng\, f = S \bullet n$$

FINITE PARTIAL FUNCTIONS AND INJECTIONS

$$[X, Y]$$

$$X \nrightarrow Y == \{f : X \rightarrow\!\!\!\!\!\!+\ Y \mid dom\, f \in \mathbb{F}\,X\}$$
$$X \rightarrowtail\!\!\!\!\!\!+\ Y == (X \nrightarrow Y) \cap (X \rightarrowtail\!\!\!\!\!\!+\ Y)$$

MAXIMUM AND MINIMUM

$$min == \lambda\, S : \mathbb{P}_1\, \mathbb{Z} \bullet \mu\, m : S \mid \forall\, n : S \bullet m \leq n \bullet m$$
$$max == \lambda\, S : \mathbb{P}_1\, \mathbb{Z} \bullet \mu\, m : S \mid \forall\, n : S \bullet m \geq n \bullet m$$

SEQUENCES

$$[X]$$

$$seq\, X == \{f : \mathbb{N} \nrightarrow X \mid dom\, f = 1\ ..\ \#f\}$$
$$seq_1 X == \{f : seq\, X \mid \#f > 0\}$$

SEQUENCE DECOMPOSITION

$$[X]$$

$$hd, last : seq_1\, X \rightarrow X$$
$$tl, front : seq_1\, X \rightarrow seq\, X$$

$$\forall\, s : seq_1 X$$
$$\bullet\ hd\, s = s\, 1$$
$$\wedge\ last\, s = s\, \#s$$
$$\wedge\ tl\, s = succ \fatsemi (2\ ..\ \#s \lhd s)$$
$$\wedge\ front\, s = 1..(\#s-1) \lhd s$$

CONCATENATION

$$[X]$$

$$(\_ \frown \_) : (seq\ X \times seq\ X) \rightarrow seq\ X$$

$$\forall s, t : seq\ X \bullet s \frown t = s \cup \{(\lambda n : \mathbb{N} \mid n > \#s \bullet n - \#s) \mathbin{⦂} t\}$$

SEQUENCE CONSTRUCTION

$$[X]$$

$$(\_\ cons\ \_) : (X \times seq\ X) \rightarrow seq\ X$$
$$(\_\ snoc\ \_) : (seq\ X \times X) \rightarrow seq\ X$$

$$\forall x : X; s : seq\ X \bullet x\ cons\ s = \langle x \rangle \frown s \land s\ snoc\ x = s \frown \langle x \rangle$$

SEQUENCE REVERSAL

$$[X]$$

$$rev : seq\ X \rightarrow seq\ X$$

$$\forall s : seq\ X \bullet rev\ s = (\lambda n : 1\ ..\ \#s \bullet \#s - n + 1) \mathbin{⦂} s$$

FILTERING

$$[X]$$

$$(\_ \upharpoonright \_) : (seq\ X \times \mathbb{P}\ X) \rightarrow seq\ X$$

$$\forall V : \mathbb{P}\ X$$
$$\bullet \langle\rangle \upharpoonright V = \langle\rangle$$
$$\land (\forall x : X \bullet (x \in V \Rightarrow \langle x \rangle \upharpoonright V = \langle x \rangle) \land (x \notin V \Rightarrow \langle x \rangle \upharpoonright V = \langle\rangle))$$
$$\land (\forall s, t : seq\ X \bullet (s \frown t) \upharpoonright V = (s \upharpoonright V) \frown (t \upharpoonright V))$$

DISTRIBUTED CONCATENATION

$$[X]$$

$$\frown / : seq\ (seq\ X) \rightarrow seq\ X$$

$$\frown / \langle\rangle = \langle\rangle$$
$$\forall s : seq\ X \bullet \frown / \langle s \rangle = s$$
$$\forall q, r : seq\ (seq\ X) \bullet \frown / (q \frown r) = (\frown / q) \frown (\frown / r)$$

DISJOINT, PARTITION

$[I, X]$

$$disjoint : \mathbb{P}(I \pfun \mathbb{P} X)$$
$$\_partition\_ : (I \pfun \mathbb{P} X) \rel \mathbb{P} X$$

$$\forall S : I \pfun \mathbb{P} X; T : \mathbb{P} X$$
$$\bullet (disjoint\ S \Leftrightarrow (\forall i, j : dom\ S \mid i \neq j \bullet (S\ i) \cap (S\ j) = \emptyset))$$
$$\land (S\ partition\ T \Leftrightarrow disjoint\ S \land \bigcup \{i : dom\ S \bullet S\ i\} = T)$$

Annex C - The Z Syntax

This syntax is given in British Standard form (BS 6154:1981) [BS] and has the following features:

1) Denotations of the terminal symbols are enclosed in quotes.
2) [and] indicate optional symbols.
3) { and } indicate repetition, that is, a possibly empty sequence of symbols.
4) Each rule has an explicit final character (a semi-colon).
5) Brackets group items together.
6) (* and *) indicate a comment.
7) A comma is the concatenate symbol, an equals sign the defining symbol and a vertical line the alternate symbol.

Named terminal symbols

id	standard identifier, including decoration
document	a specification module
decor	?, !, ′, or subscript
EZ	end of Z picture
NL	hard new line
SI	start indentation
EI	end indentation
finish	end of file
SR	start vertical rule
ER	end vertical rule
UD	unique (generic) definition
inset	infixed generic sets
preset	prefixed generic sets
postset	postfixed generic sets
op	infix operator
encop	lhs of enclosed operator
distinop	lhs of distributed infix operator
distpreop	lhs of distributed prefix operator
eop	delimiter of two part operators
preop	prefix operator
postop	postfix operator
sconst	numbers and the constants
TH	start theorem
ETH	end theorem
SW	where phrase delimiter
where	where phrase delimiter
EW	where phrase delimiter
rel	relational operator
explicit_set	{ when indicating start of set display
pre	pre-condition schema operator
SB	start schema box (after name)

ST	middle line of schema box
ESB	end schema box

Rules

```
z_text = [z_phrase], finish
       | z_phrase, z_sep, z_text;

z_sep = list_sep | EZ;

list_sep = ';' | NL;

z_phrase = given_set_def
         | definition
         | constraint
         | theorem
         | import
         | export;

given_set_def = '[' , given_ids, ']';

given_ids = id, { ',' , id };

definition = axiomatic_def
           | syntactic_def
           | datatype_def
           | schema_def;

constraint = pred;

theorem = '⊢' , pred
        | TH , [gen_params], [hyps], '⊢' , pred_list, ETH;

hyps = hyp, {list_sep, hyp};

hyp = pred | dec | pred, '|' , dec;

import = doc, {doc};

doc = document, [ decor ], [instantiation];

export = id, 'keep', idslist;

ids = id | inset | preset | postset | op | rel | encop, eop
    | distinop , eop | distpreop , eop | preop | postop;

idslist = ids, { ',' , ids };
```

```
reference = ( id | id , '$' , id ), [instantiation];

instantiation = '[' , inst_list, ']' ;

inst_list = inst_term_list | binding_list | rename_list;

(* The two forms of instantiation and schema renaming are all treated as
instantiation in this syntax: the various possibilities are distinguished semantically
*)

inst_term_list = term, { ',' , term };

binding_list = id, '=' , term, { ',' , id, '=' , term };

rename_list = id, '/', id, {',', id, '/', id };

axiomatic_def = liberal_def
              | unique_def
              | generic_def;

liberal_def = dec, [ '|' , pred ]
            | SR, def_body, ER;

def_body = dec_list, [ ST , pred_list ];

unique_def = UD, def_body, ER;

generic_def = decl_name, gen_params, ':' , term, '|' , pred
            | UD, gen_params, def_body, ER ;

gen_params = '[' , given_ids, ']' ;

dec_list = dec, {list_sep, dec};

dec = decl_name, {list_sep, decl_name}, ':' , term;

(* decl_name has options to indicate the syntactic status of the operator being defined *)

decl_name = id | id , '_' | '_' , id | id , '_' , id
          | '(' , '_' , id, '_' , ')'
          | '_' , id , '_'
          | id, '_' , id , '_'
          | '_' , id, '_' , id;

syntactic_def = syn_def_id
              | decl_name, gen_params, '==' , term
              | UD, gen_params, syn_def_list, ER;
```

```
syn_def_id = decl_name, '==', term;

syn_def_list = syn_def, {list_sep, syn_def};

syn_def = syn_def_id | syn_def_ids;

(* syn_def_ids defines prefix, postfix and infix generic sets *)

syn_def_ids = id, id, '==', term
            | id, id, id, '==', term;

datatype_def = id, '::=', branch, { '|', branch };

branch = id | id, '«', term, '»';

schema_def = id, [gen_params], schema_definition;

schema_definition = '≙', schema_term | schema;

schema = SB, local_dec_list, [ ST, pred_list ], ESB;

pred_list = pred, {list_sep, pred};

local_dec_list = (dec | inclusion), [list_sep, local_dec_list];

inclusion = ( id | id, '$', id), [instantiation];

(* Only schema renaming is allowed here *)

explicit_constr = tuple
                | explicit_set, '}'
                | explicit_set, termlist1, '}'
                | '⟨', [termlist1], '⟩';

termlist1 = term, { ',', term };

tuple = '(', term, ',', termlist2, ')'
      | 'θ', reference;

termlist2 = term, { ',', term };

(* aform = atomicformulae, and includes predicates. The distinction between terms
and predicate is checked semantically *)

aform = 'ℤ' | 'Char' | sconst
      | reference
      | aform, '.', id
```

```
        | '(' , product, ')'
        | explicit_constr
        | '{' , local_dec_list, [ '|' , pred ], [ '•' , term ], '}'
        | '(' , partials, ')'
        | encop, term, eop
        | SW, ax_dec_list, where, pred_list, EW
        | SW, syn_def_list, where, pred_list, EW
        | '(' , pred, ')' ;

product = term, '×', term, { '×', term };

ax_dec_list = local_dec_list, '|' , pred
            | SR, dec_list, ST, pred_list, ER;

(* partials corresponds to various forms of partial application *)

partials = '_' , rel, '_'
        | '_' , op, form2
        | aform, op, '_'
        | '_' , op, '_'
        | '_' , distinop, term, eop
        | aform, distinop, '_' , eop
        | '_' , distinop, '_' , eop
        | distpreop, '_' , eop, '_'
        | distpreop, term, eop, '_'
        | distpreop, '_' , eop, form3
        | encop, '_' , eop
        | preop, '_'
        | '_' , postop;

(* The following syntax rules define the priority of the various operator symbols. The
weakest binding is the infixed generic sets such as → *)

formula = form1, {inset, form1};

(* infixed operators *)

form1 = [ form1, op ], form2;

(* function application *)

form2 = [form2], form3;

(* prefix function application and generic sets *)

form3 = preop, form3
        | preset, form3
```

```
        | distpreop, term, eop, form3
        | 'ℙ', form3
        | form4;

(* postfix function application and generic sets *)

form4 = form4, distinop, term, eop
        | form4, postop
        | form4, postset
        | aform;

comprehension = 'λ', local_dec_list, lambda_set
                | 'μ', local_dec_list, lambda_set;

lambda_set = [ '|', pred ], '•', term;

term = comprehension | formula;

apred = SI , pred_list, EI | term;

rel_exp = term, '∈', term
          | term, '=', term, [tail]
          | term, rel, term, [tail]
          | apred;

tail = rel, term, [tail]
     | '=', term, [tail];

(* priority of connectives: *)

log_exp = log_exp1 | log_exp, '⇔', log_exp1;

log_exp1 = log_exp2 | log_exp1, '⇒', log_exp2;

log_exp2 = log_exp3 | log_exp2, '∨', log_exp3;

log_exp3 = log_exp4 | log_exp3, '∧', log_exp4;

log_exp4 = { '¬' }, rel_exp;

quant_exp = quant, dec_list, [ '|', pred ], '•', pred;

quant = '∃', '∃1', '∀';

pred = quant_exp | log_exp;

schema_term = quant_sexp | log_sexp;
```

```
quant_sexp = quant, dec_list, '•', schema_term;

log_sexp = log_sexp1 | log_sexp, '⇔', log_sexp1;

log_sexp1 = log_sexp2 | log_sexp2, '⇒', log_sexp1;

log_sexp2 = log_sexp3 | log_sexp2, '∨', log_sexp3;

log_sexp3 = log_sexp4 | log_sexp3, '∧', log_sexp4;

log_sexp4 = spec_sexp | '¬', log_sexp4;

spec_sexp = spec_sexp, '/', '(' , id_list, ')'
          | spec_sexp, '/', reference
          | spec_sexp, 'op', spec_sexp1
          | spec_sexp, '»', spec_sexp1
          | spec_sexp1;

id_list = id, {',', id};

spec_sexp1 = [ 'pre' ], spec_sexp2;

rename = '[' , rename_list, ']' | decor;

spec_sexp2 = '(' , schema_term, ')', [rename]
           | reference
           | schema;
```

Reference

[BS] "Method of defining syntactic metalanguage", British Standards Institution, BS 6154:1981

DOCUMENT CONTROL SHEET

Overall security classification of sheet ..... Unclassified .....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification, eg (R), (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	Memo 4356		Unclassified

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
7784000	ROYAL SIGNALS & RADAR ESTABLISHMENT ST ANDREWS ROAD, GREAT MALVERN WORCS WR14 3PS

5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title

ZADOK USER GUIDE

7a. Title in Foreign Language (in the case of Translations)

7b. Presented at (for Conference Papers). Title, Place and Date of Conference

8. Author 1. Surname, Initials	9a. Author 2	9b. Authors 3, 4 . . .	10. Date	pp. ref
RANDELL G P			1990.01	34

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution Statement

Unlimited

Descriptors (or Keywords)

Continue on separate piece of paper

Abstract

This is a guide for users of ZADOK, the RSRE Z syntax and typechecker. It also contains a brief introduction to using the Perq Flex system.