Project:	FST PROJECT		
Title:	Issues for Z Concrete Syntax		
Ref:	DS/FMU/IED/WRK036	*Issue:* 1.7	*Date:* 28 February 1992
Status:	Informal		*Type:* Report
Keywords:	HOL		

Author:

Name	*Location*	*Signature*	*Date*
R.D. Arthan	WIN01		

Abstract: A discussion of issues in the design of a concrete syntax for Z, including a complete proposal for the concrete syntax.

Distribution: Library

0 DOCUMENT CONTROL

0.1 Contents List

0.2 Document Cross References

[1] S. King, I.H. Sorensen, and J. Woodcock. Z: Grammar and Concrete and Abstract Syntaxes. *Programming Research Group, University of Oxford*, 1987.

[2] C. T. Sennett. *Syntax and lexis of the specification language Z.* RSRE Memorandum 4367. MOD PE, RSRE, February 1990.

[3] J.M. Spivey. *Understanding Z.* Cambridge University Press, 1988.

[4] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 1989.

[5] BS6154:1981. *Method of defining syntactic metalanguage.* British Standards Institution, 1981.

[6] *Lexical issues in the standardisation of Z.* C. T. Sennett, Defence Research Agency, Malvern, 1992.

[7] ZIP/PRG/91/074. *Z Base Standard (version 0.4).* Z Standards Change Group, Oxford University Programming Research Group, 9th December 1991.

0.3 Changes History

Issues 1.2-1.6 (5-19 February 1992) Internal Drafts.

Issue 1.7 (28 February 1992) First issue for distribution outside ICL.

0.4 Changes Forecast

It may be useful to propose a description of an extended character set as an example of one way of encoding a sufficient set of symbols as 8-bit bytes.

1 GENERAL

1.1 Scope

This paper discusses some issues in the design of a concrete syntax for Z. It is intended as input to the work on this topic by the Z standards committee. An appendix contains a complete proposal for a possible concrete syntax.

1.2 Introduction

The concrete syntax for Z is a potential cause of endless debate. Despite several attempts, [4, 2, 1], there is currently no formulation of the syntax sufficiently precise to establish exactly what is and is not allowed. Moreover, discussions and documents arising in the standardisation process have shown rather disparate views of how radically the new standard should differ from what has gone before.

This document is an attempt to establish some criteria by which competing proposals may be judged and to apply them to some of the issues of which the author is aware. Note that I am concerned here with the language to be standardised and not with the formalisms to be used to specify it or with the style of their specification.

There would seem to be four main attributes which the design of the concrete syntax for Z will affect. In order of decreasing importance these are as follows.

Ease of Understanding(UND) — the ability to express a concept clearly;

Ease of Expression(EXP) — the ability to express a concept concisely;

Ease of Conversion (CNV) — the ability to transfer specifications from earlier versions of Z to the standard language;

Ease of Implementation(IMP) — the amenability of the syntax to mechanical checking and manipulation

The way in which these attributes trade against each other is quite complex. To simplify evaluation against these attributes we will take it that *UND* and *EXP* are to be measured in terms of the relationship between concrete and abstract syntax, i.e., we shall consider a proposal to supply *EXP* if it enables a class of abstract syntax constructs to be expressed easily and concisely and to supply *UND* if it enables such a class to be expressed clearly.

An important aspect of *UND*, which might be added to the above list of attributes is ease of learning and remembering the language. Very often in real-life applications a specification will need to be read by people with only a small knowledge of Z. It is important at the very least that such a reader should know which parts of the specification he does and does not understand and why.

Assessing *IMP* is not easy without assuming some rather specific details of what tools are expected to do and how they are to be implemented. We take the view *IMP* should be assessed on the assumption that the tools in question will include a parser and type-checker and a pretty printer (i.e. they will support translation from concrete syntax to abstract and vice versa). A measure of the complexity of parsing and type-checking which we shall occasionally use is the complexity of the data structure required to record contextual information in the analysis of a fragment of Z. This measure also bears upon *UND*, since someone reading a specification also has to compute and remember this contextual information.

To add a semblance of scientific verisimilitude and to provide material for debate, specific proposals discussed in this document are assigned mnemonic names. I have marked each proposal against each of the four attributes. These marks are, of course, based on personal opinions, but reasons are usually given. The marks range from -2 to 2 and are intended to be interpreted as shown in the following table:

2	Significant benefit
1	Some benefit
0	No impact
-1	Some adverse impact
-2	Significant adverse impact

CNV has been marked in terms of conversion of a specification conforming to [4]. When a proposal amounts to 'do the same as in [4]', the mark for *CNV* is given as $-$.

1.3 Abstract Syntax

To simplify the discussion, this document is mostly based on the supposition that the abstract syntax for Z is as in the draft standard (version 0.4) with the modification that the notion of paragraph (*PAR*) in section A.2 be changed to read something like the following:

Modified Abstract Syntax

$$\begin{array}{lcl} PAR & = & GIVENSETDEF \\ & | & AXDEF \\ & | & ABDEF \\ & | & FREETY \\ & | & GLOBALPRED \\ & | & SCHEMADEF \\ & | & GENERICDEF \end{array}$$

in which the new categories *AXDEF*, *ABDEF*, and *FREETY* correspond directly to axiomatic definitions, abbreviation definitions and free type definitions and are defined as follows:

Modified Abstract Syntax

$$\begin{array}{lcl} AXDEF & = & let\ SCHEMA \\ ABDEF & = & WORD[WORD,...,WORD] == EXP \\ FREETY & = & WORD ::= BRANCH \mid ... \mid BRANCH \\ BRANCH & = & WORD \langle\langle\ EXP\ \rangle\rangle \end{array}$$

In fact we shall argue that *SCHEMADEF* and *ABDEF* may be merged as may *AXDEF* and *GENERICDEF* (see section 10.3 below).

Some of the proposed concrete syntax we consider, mainly related to local definitions (i.e., *let*-expressions and *let*-predicates) and operations on bindings require further modifications to the abstract syntax. These will be described as necessary.

1.4 Context Conditions

The existing Z documentation is not fully rigorous on the subject of context conditions (aka. “static semantics” or “well-formedness rules” or “type checking and scope rules” or . . .). This weakness has led to a great deal of confusion and has made some notions in Z, particularly to do with schemas, rather hard to grasp, even for those who are mathematically competent. Moreover, even the cognoscenti disagree in their interpretations of what the rules implicit in the existing documentation really are.

The actual details of the context conditions have a significant impact on the concrete syntax (since they influence the information which can be inferred from the context in which a construct appears and does not, therefore, have to be explicitly stated by the user).

We shall discuss some issues relating to context conditions as they arise, although I make no claim that a unified account is given here. In particular, the typing rules are not addressed here.

2 LEXICAL ANALYSIS

2.1 General

Neither [4] or [1] really address this issue. [2] and a recent working note, [6], give some proposals and I have only minor disagreements with the way these treat the splitting up of a sequence of "characters" into lexical tokens. In particular, I approve of the way that [6] does not limit the language to a fixed character set. I prefer to separate higher level questions such as how names are classified as infix operators etc. from the more primitive aspects of lexical analysis (such as deciding what is a name) and, since such questions do merit more careful discussion, later sections of this document discuss them in more detail.

2.2 Layout

One lexical question is whether or not Z should be a "free format language", allowing free use of spaces tabs and line breaks, in the way that nearly all modern programming languages are. Somewhat surprisingly, [4], very explicitly says that line breaks are significant and lists the places where they are allowed (and, presumably, they are not allowed in other places). This would seem to me to be unacceptable for real-life use of Z — the (questionable) benefit of being allowed to omit ';'s in predicates and declarations does not merit the disadvantage of not allowing the user to format a long or complex expression in a readable way. Indeed, the rules of [4] effectively prohibit the use of meaningful variable names in some cases, by implying a fixed limit (of about half the number of identifiers which will fit on a line) on the number of actual parameters which may be supplied in an iterated function application.

Calling the proposal to use the rules of [4] for the use of line breaks, *Layout.1*, and a proposal allowing free use of line breaks *Layout.2* by asking the user to use ';'s where it is not possible to give a convenient syntax making them optional, the merits of the two proposals are shown in the following table:

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
Layout.1	Line breaks only allowed in some positions	*−2*	*−2*	-	*0*
Layout.2	Free use of line breaks allowed	*2*	*2*	*−1*	*0*

Subsequent proposals will endeavour to be compatible with *Layout.2*.

3 NAMES

The topic of names is as complicated as any.

3.1 Decorations in Schema-as-Expression

[4] prohibits the decoration of schema references when used as an expression. I know of no reason for making this restriction: it would seem quite in order, and useful, for S' to denote what you get from the set of bindings S by decorating each name in each binding with $'$.

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
DecSchRef.**1**	Decoration of schema-as-expression banned	-2	-2	-	*0*
DecSchRef.**2**	Decoration of schema-as-expression allowed	*2*	*2*	*0*	*0*

3.2 Recognising Decoration

[4] distinguishes two classes of names, schema references and identifiers. The two sorts of name behave differently when they are followed by decoration characters. What is not made clear is the context conditions which enable one to distinguish between the two sorts of name. Presumably, the distinction is to be made on the basis of whether or not the name was declared using a schema box (or equivalently, a horizontal schema definition). We will call this scheme *Decor.1*. To make *Decor.1* work, names declared in schema boxes are not allowed to include decoration.

While the problem it solves is actually rather an obscure one, and while most specifications are not affected by it, *Decor.1* does not seem to be the best way of tidying up the loose end. It is bad for *EXP* and *UND*, because it can happen that two named objects are provably equal but are not interchangeable. E.g. consider the following fragment of specification:

A

$a : \mathbb{Z}$

---

true

$B : \mathbb{P}\ A$

$B = A$

Clearly this is consistent and allows one to prove $A = B$. However, A' and B' are quite different under *Decor.1*. *Decor.1* is also not particularly good for *IMP* since it forces the parser to keep information about what sort of paragraph was used to declare which names in its context.

An alternative solution to the problem of identifying when decoration really is decoration based solely on knowledge of which variables are in scope is given by the following rules (*Decor.2*):

1. We are given a *Word*, w, immediately followed by a *Decoration*, d. We have to decide whether this construct is a decorated schema name or whether $w \frown d$ is just to be treated as a plain

identifier. If the construct is appearing to the left of a ':' in a declaration then we always treat it as a plain identifier (this is a declaring instance of the identifier).

2. Otherwise this is an applied instance of an identifier. Using our knowledge of the variables in scope, we find the longest prefix, c say, of d such that $w ^\frown c$ is in scope. If no such c exists, then take $c = d$.

3. If $c = d$, then there is no way of interpreting the construct as a decorated schema name and so we treat $w ^\frown d$ as a plain identifier.

4. Otherwise we treat the construct as the schema name $w ^\frown c$ decorated with the strokes obtained by removing c from the beginning of d.

Note that in stage 4 above, if $w ^\frown c$ does not have the right type to be a schema then the construct will be taken to be ill-typed. (It is conceivable that $w ^\frown d$, could have a legal interpretation if some prefix e of c is such that $w ^\frown e$ has the right type to be ae a schema. Stage 2 above could be modified to find this e, but the above rules are felt to be easier for a human reader of a specification.)

Decor.2 has the advantage of being backwards compatible with any reasonable interpretation of [4] (for which the context conditions must, surely[1], guarantee that c above is either empty or is equal to d).

A third proposal in this area, *Decor.3*, conceptually even simpler than *Decor.2*, is just to demand that a *Word* must be separated from its *Decoration* by some white space. *Decor.3* has the slight disadvantage over *Decor.2* in terms of *CNV*, since one needs to know which names are schemas and which are not to insert the spaces in an old specification. *Decor.3* is likely to be slightly easier to implement than *Decor.2* since it can be done during parsing rather than during type checking (since one needs to know the types of schemas in order to establish what is in scope).

The following table shows the estimated merits of the three proposals:

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
Decor.1	Names of objects defined with schema box are special with respect to decoration	*−2*	*−2*	-	*−1*
Decor.2	Decoration recognised on basis of scope information	*2*	*2*	*2*	*1*
Decor.3	Decoration distinguished by use of spaces	*2*	*2*	*−1*	*2*

3.3 Use of Brackets in Names

In [4], a distinction is made between *VarName*s and *DecName*s. The difference is that in the means for suppressing the special status of an operator (as infix, prefix, or postfix). In a *VarName*, brackets are required around such operators, whereas in a *DecName* they are not. It might be better to ask for the brackets under all circumstances, since otherwise the user has to worry about when the brackets are allowed.

[1] It is, in fact, not clear from [4] whether in the scope of a schema, named S, say, it is legal to declare locally a variable named S'. If it is, then in the scope of the local declaration, S' must surely refer to the local variable rather than S.

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
VrDcNm.1	Brackets required around operators used as names.	*2*	*2*	-1	*2*

3.4 Overloaded Identifiers

In [4], the identifiers '$\upharpoonright$' and '$\fatsemi$' are used both for language primitives and as global variables in the mathematical toolkit. This overloading of these symbols has little benefit to the reader or writer of a specification and is a source of technical difficulty both in specifying the language and in implementing tools.

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
Ovrld.1	Do not let toolkit identifiers overload language primitives.	*2*	*2*	-1	*2*

4 DECLARATIONS

4.1 Fixity

As discussed in [2], references [1] and [4] involve between 3 and 6 different possible fixities a name can have and three sorts of name which can possess such fixity. Not all combinations are offered in either [1] or [4]. [1] gives what would seem to be a workable scheme for deducing the syntactic status from the form of a declaration (although the scheme relies on a rather unintuitive use of brackets to distinguish functions from relations). [4], however, takes the following position:

> "Some infix, prefix and postfix symbols are standard; ... Others may be introduced as they are needed, but each symbol should be used consistently throughout a document. Some specification tools work with a table of symbols which can be extended, but there is no standard way of doing this."

I.e. you can introduce infix symbols etc. as needed, despite the fact that [4] appears to give you no way of doing this and suggests that not every specification tool may support it. (Not surprisingly, many readers of [4] infer from the BNF description it gives that it is offering as similar scheme to [1], but this would not seem to be the intention.)

The six fixities in the cited references do not include a facility for the user to define outfix notations like the bag display brackets which operate on sequences. Including such a possibility in an orthogonal way we arrive at the nine forms shown, with examples, in the following table:

Infix	$\_+\_ \quad \_\cup\_$
PreFix	$hd\_$
PostFix	$\_^{-1}$
OutFix	$\lbag\_\rbag$
OutFixPre	$\lbag\_\rbag\_$
OutFixPost	$\_\lbag\_\rbag \quad \_^{-}$
OutFixSeq	$\lbag\_\ldots\rbag$
OutFixSeqPre	$\lbag\_\ldots\rbag\_$
OutFixSeqPost	$\_\lbag\_\ldots\rbag$

(Here I take it that the operation of superscription may be treated as an OutFixPost. X^Y is taken to be the printed appearance of something like $X \nearrow Y \updownarrow$, the characters $\nearrow$ and $\updownarrow$ delimit what is to be superscripted.)

There are three sorts of language construct which may be denoted by using the special syntax associated with a fixity:

Function Application	*InFun*, *PreFun* etc. in [4]
Membership Assertion	*InRel*, *PreRel* etc. in [4]
Generic Instantiation	*InGen*, *PreGen* etc. in [4]

The mnemonics, *fun*, *rel* and *gen* may conveniently be used for the three sorts of construct.

Since functions are sets and may be generic and since instances of generic constants may also be functions or sets, the distinction between these three sorts of construct cannot be settled on the basis of type information alone. For example, consider the generic definition

$$[X, Y]$$

$$\_\ fst\ \_ : X \times Y \to X$$

$$\forall x{:}X;\ y{:}Y \bullet x\ fst\ y = x$$

In principle, for a and b of types A and B respectively, any of the following make sense:

Example

$$\begin{array}{lcl} a\ fst\ b & = & (\_fst\_)(a,\ b) \\ (a,\ b)\ fst\ a & \Leftrightarrow & ((a,\ b),\ a) \in (\_fst\_) \\ (A\ fst\ B)(a,\ b) & = & (\_fst\_)[A,\ B](a,\ b) \end{array}$$

The approach taken in [4] is to restrict attention to a limited set of combinations of syntactic status and sort of name, for which, presumably, it is possible to infer the intended fixity and sort from the form of the declaration. This entails abandoning the outfix forms (so that relational image, bag display, and iteration require special treatment in the concrete syntax) and not allowing some of the remaining combinations (e.g. postfix generics).

[1] allows most of the combinations (but does not have the sequence versions of the outfix forms) and implies that all the necessary information can be inferred from the form of a declaration. The rules are to be that functions are distinguished from sets by using brackets around the identifier in the declaration and that generic constants with special syntactic status are the ones defined using syntactic definitions (i.e. abbreviation definitions in the terminology of [4]). [1] also makes the restriction that these conventions are only supported for declarations of global variables.

Another aspect of this topic is the relative precedence of names with special syntactic status. [1] explicitly leaves this issue undefined. [4] discusses it and gives the precedences of the infix functions used in the mathematical toolkit, but gives no mechanism for the user to assign precedences to names. One approach to this problem is simply not to have a scheme of operator precedence, a solution which would very probably be better than the particular assignment of precedences for toolkit functions given in [4], which seems to be have somewhat surprising consequences, e.g.:

Example

$$\begin{array}{lcl} A \cup B \lhd f \oplus C \cap D \lhd g & = & A \cup (((B \lhd f) \oplus C) \cap (D \lhd g)) \\ 1 \mathrel{..} 10 \frown 20 \mathrel{..} 30 & = & (1 \mathrel{..} (10 \frown 20)) \mathrel{..} 30 \end{array}$$

However, an operator precedence scheme used sparingly and with care can help greatly in eliminating irrelevant brackets, e.g. one really shouldn't have to put any brackets in $1 \mathrel{..} 10 \frown 20 \mathrel{..} 30$ since the types of the operators involved ensures that there is only one sensible way to bracket the term. Specification tools can easily help in this area both by allowing expressions to be displayed with bracketing or spacing to show what is going on and also by warning the user of the possible bracketing ambiguities when an infix or other special operator is declared.

Some specification tools allow the user to set operator status and precedence by means outside Z. One approach, which makes Z work much more like ordinary mathematical usage (in which one expects things like $+$ or $\cup$ or $\otimes$ to be infix operators regardless of context), is to have a new form of paragraph in which fixity may be defined as a property of names rather than variables. This is very like the treatment of infix operators in Standard ML. An example might be:

Example

fun	*10*	$\_\frown\_$
fun	*20*	$\_..\_, \_\llparenthesis\_\rrparenthesis$
rel	*10*	$\_\notin\_$
rel	*10*	$\_\neq\_$
gen	*30*	$\_\pfun\_, \_\ffun\_$

to indicate that $\frown$, .. etc. are to be used as operators of the given sort, precedence and fixity throughout the specification. It would of course then be sensible to make it illegal (or at least to expect tools to warn the user) if such a fixity paragraph for a name was incompatible with its declared status as *op* or *rel* or *gen*. I would suggest that it be allowed to have several fixity paragraphs for the same name but that all such paragraphs should give the same status to the name (so supporting a style in which the fixity information for a specification may be collected together into one ready-reference section and may also be repeated as names are used where that is thought to be a useful reminder).

We thus arrive, so far, at (approximately) three proposals for the treatment of fixity (in fact, there are a lot of possible variants on the fine details).

Fixity.1 A working out of the scheme in which the form of a global or local declaration is used to infer the fixity of the variables being declared (from a limited subset of the above-mentioned fixities and sorts). This will require special syntax (e.g., the brackets used in [1]) to distinguish functions from relations and some conventions about which forms of generic definition declare generic infixes etc. This scheme means that the toolkit notations for relational image and sequence and bag display have to be treated as special cases and that user-defined infixes etc. will all have to have the same precedence (as will the toolkit operators if they are not also to be special cases)

Fixity.2 The scheme of [1] extended to allow prefix relations and the various sequence outfix forms. This has the same problems as *Fixity.1* as regards user-defined precedence but does cater for all the fixities used in the mathematical toolkit.

Fixity.3 The scheme sketched above in which a fixed range of sort, precedence and fixity combinations are assigned as global properties of names in a new form of paragraph.

It is intended in *Fixity.1* that the the form of a declaration inside a schema box should not affect contexts in the scope of uses of the schema as a declaration. Thus for example given

S

$\_x\_ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$

S

...

x does not have infix status in the predicate part of the axiomatic box. This is *(a)* to avoid problems with *UND* and *IMP* and *(b)* to give a simple treatment of the difficulties inherent when we use a schema such as S above in a schema expression as in:

```
┌─T──────────────────────────
│   x_:ℤ × ℤ → ℤ
└────────────────────────────

│   U ≙ S ∨ T
```

Note that this problem does not arise with the other proposals.

More general than any of the above would be schemes in which a much larger class of user-defined notations would be permitted, e.g. allowing a function application, say $R\,x$, to be used as with infix syntax, as in $a\ R\ x\ b = (\_R\ x\_)\ (a,\ b)$. A scheme of this sort as has been proposed by S. Stepney. This would seem to me to be a research topic on too large a scale for assessment at the time of writing. One can certainly envisage such a scheme being more readily added to *Fixity.3* than to either of the other two schemes. A modest move in this direction would be to generalise (and simplify) *Fixity.3* to allow arbitrary mixfix notations, including the forms offered by *Fixity.3* as special cases. This would allow fixity declarations like the following as well:

Example

```
│   fun   10   if _ then _ else _
│   fun   10   case _ in ... esac
```

Thus the relevant part of the syntax of expressions becomes something like:

Example BNF

```
│   E   =   ...
│       |   [E], op, {E, {‘,‘, E}, op}, [E]
│       |   ...;
```

This gives a fourth proposal for the fixity problem:

Fixity.4 Like *Fixity.3* but generalised to allow arbitrarily long mixfix notations.

The following table gives the estimated merits of the three proposals:

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
Fixity.1	Similar to [4]	*−1*	*−1*	*0*	*−1*
Fixity.2	Similar to [1]	*1*	*1*	*0*	*0*
Fixity.3	Fixity declarations for fixed set of forms given separately	*2*	*1*	*0*	*2*
Fixity.4	Fixity declarations for general mixfix forms given separately	*2*	*2*	*0*	*2*

Here *Fixity.1* is viewed as harder to implement than the others since it means that fixity has to be passed around as an attribute synthesised from local declarations, whereas in the other two schemes it is ascertainable from global declarations (which for *Fixity.3* are of a very simple form). *Fixity.3* is considered to offer best *UND* and *EXP* since it is felt to be closer both to ordinary mathematical usage (in which symbols such as $+$, $\otimes$ etc. are used as infix operators regardless of their intended meaning) and to programming languages such as Standard ML.

4.2 Association of Operators

Prefix (resp. postfix) operator necessarily associate to the right (resp. left) since, e.g., $---a$ just cannot be bracketed as $((-)-)-a$.

In [4], left and right association is stated for different sorts of infix names and for the logical connectives. The use of a fixed association seems sensible (since it would complicate any of the schemes discussed in the previous section to make the association user-definable). However, there seems to be no point in not just fixing on right association for everything. This makes no difference for the logical connectives, since all of them except implication are associative, and that is most naturally taken as right associative. Moreover, most infix functions are either associative or are naturally best as right associative (e.g. one usually wants $a \mapsto (b \mapsto c)$ rather than $(a \mapsto b) \mapsto c$). The only toolkit operations which would be adversely affected are, I believe, range restriction and anti-restriction. Of the schema relevant calculus infix operators[2], only projection, $\upharpoonright$, is not associative. On balance, therefore, right association everywhere seems a good rule.

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
Assoc.**1**	Association rules as per [4]	*−2*	*−2*	–	*0*
Assoc.**2**	All infix expression constructs and the logical infix schema operators right associative	*2*	*2*	*0*	*0*

4.3 Interplay with Fixity

If either of the proposals *Fixity.1* or *Fixity.2* mentioned above (or indeed any scheme in which fixity is derived from the form of a declaration) is adopted, then there is an interplay between the treatment of fixity and between the treatment of scope in declarations. For example, consider a declaration in which a variable appears on both sides of a ':', e.g.:

Example

$$x : \mathbb{P}y;\ z : \mathbb{P}x;\ x : \mathbb{P}x$$

If this is illegal as in [4], there is no problem for the concrete syntax, otherwise we must arrange for all the applied instances x to have the fixity implicit in the appropriate declaration and this will depend on the precise details of the scope rules chosen.

4.4 Separating Declarations

[4] and [1] both allow declarations to be separated either by ';'s or line breaks. As suggested in section 2.2 above, it seems bad for line breaks to have special significance. This seems to lead to three possible proposals:

DecSep.1 to insist on a ';' between declarations.

DecSep.2 to allow the ';' between two declarations to be replaced by white space if it is clear from the context where the ';' must go.

[2] I do not consider schema hiding to be a binary infix operator of the sort which requires an association rule. It is necessarily 'left associative', in some sense, since the 'right associative' interpretation is impossible.

DecSep.3 to allow the ';' to be omitted by restricting the expression which appears after the ':' to be one of the "enclosed" forms of expression whose end point is easily specified by a grammar (effectively this requires expressions like $A{\times}B$ to be enclosed in brackets when used after a ':', if the ';' is to be omitted).

The problem in *DecSep.2* is caused by the fact that a declaration is either a comma-separated list of names followed by a followed by an expression, or a schema reference. So for example x, y, $z{:}A\ B$ can, depending on the context, be taken as equivalent to x, y, $z{:}A$; B or x, y, $z{:}(A\ B)$. Thus in *DecSep.2* one would have to base the decision on where to put the ';' on the types of A and B. This seems rather complex.

The following table gives the estimated merits of the three proposals:

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
DecSep.1	';' mandatory between declarations	*2*	*2*	*−1*	*2*
DecSep.2	';' between declarations may be elided (context dependent)	*−2*	*−2*	*1*	*−2*
DecSep.3	';' between declarations may be elided (but brackets needed instead sometimes)	*1*	*−1*	*−1*	*0*

4.5 Elision of Set Constraints

The genericity in Z is in many ways similar to the polymorphism in languages such as ML and Miranda, which have polymorphic type systems supported by Milner's type inference algorithm. In fact, the treatment of generics in [4] forces a type checker for Z to go to most, if not all, of the trouble of doing full type inference, but the language does not exploit this by allowing users to omit type information in declarations. Stylistically, it is probably a good thing for global declarations to be type-constrained (I should actually say set-constrained), although it is worth observing that there is no way for the user to put in useful type information in abbreviation definitions or schema definitions).

However, it can be quite a nuisance both to read and to write specifications in which you have to supply type information in local declarations which is both complicated and uninteresting. It is infuriating in such cases to use a type checker which you know is just working out what the types are, and then complaining if you worked them out differently!

To see that the language of [4] does indeed support full type inference to all intents and purposes, consider the generic definition:

$$U[X] == X$$

In the scope of this one can write things like:

$$f : U$$
$$n : U$$

$$n > 0;$$
$$\forall x{:}U \bullet f\ x \leq n$$

However, the use of the user-defined object U here (or perhaps something called *Universe* or *Totality* in the mathematical toolkit), is not very perspicacious. It would look better if the user could just omit the set constraints in local declarations. Thus we have the following proposal:

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
DecEliSet.1	The term after the ':' may be omitted in a local declaration	*2*	*1*	*2*	*1*

4.6 Elision of Generic Actuals in Schema References

[4] makes it mandatory to supply the generic actual parameters in a reference for a generic schema except within a θ-expression. There seems to be no reason not to allow the generic actuals to be elided in any context in which the types can be uniquely determined as with generic constants.

5 EXPRESSIONS

There are a few isolated issues with the expression syntax of [4]. We note in passing that the expression syntax can be simplified if we adopt either of the proposals *Fixity.2* or *Fixity.3* above which mean that the notations for relational image, sequence display, bag display iteration are supported by a general facility rather than being special cases.

5.1 Unary Negation

Unary negation (of integers) is a special case in the grammar given in [4]. This allows the minus sign to be used both infix and prefix. It might be felt neater to follow the Standard ML approach and use different symbols for the two operations.

Thus:

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
UnNeg.1	Unary negation handled as special case	*1*	*2*	–	*−1*
UnNeg.1	Unary negation called '∼' rather than '-'.	*2*	*2*	*−1*	*2*

5.2 Singleton Set Displays

[4] and [1] agree that if S is a schema reference then $\{S\}$ should be a set comprehension rather than a singleton set display, and that the set display must be written $\{(S)\}$. This seems to me to be go against normal mathematical usage (in which, *(i)* a single variable name within braces would be expected to denote a singleton set, and, *(ii)* brackets around an expression are not expected to change its value). As the set comprehension is equal to S you can just write S if that's what you want, and if for some reason you really want write a comprehension you can write $\{S|true\}$.

Thus:

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
SingSetDisp.1	If S is a schema reference then $\{S\}$ is a set comprehension	*−2*	*−2*	*0*	*0*
SingSetDisp.2	If S is a schema reference then $\{S\}$ is a set display.	*2*	*2*	*0*	*0*

5.3 Scope of λ and μ

In effect, the grammar given in [4] means that one must always put brackets around a λ- or μ-expression. An alternative proposal is that such expressions should not always need brackets but rather that the expression should extend as far to the right as possible. Note that this second proposal must allow brackets to be omitted in $\lambda x{:}X \bullet x = \lambda y{:}X \bullet y$, since the first expression could not include the equals symbol (which is not allowed within an expression).

There is not much to choose between these two proposals:

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
LamMuScp.1	Brackets required around λ- and μ-expressions	*2*	*1*	*0*	*0*
LamMuScp.2	λ- and μ-expressions extend as far right as possible	*1*	*2*	-1	*0*

5.4 Schema Expressions as Expressions

Since the abstract syntax permits it we must certainly allow arbitrary schema expressions to be used as expressions. It seems best to allow decoration of such expressions.

5.5 Local Definitions

[1] offers a local definition facility by supporting where clauses in predicates. The local definitions in the where clause are either abbreviation definitions (i.e. just equations) or axiomatic definitions. It would also be desirable to have local definitions within expressions. However, for expressions there are semantic problems in allowing anything other than abbreviation definitions as local definitions. Moreover, the meaning of the predicate form is somewhat obscure when a loose axiomatic definition is used (and if the axiomatic definition is not loose a μ-expression may be used to give an equivalent, and clearer, abbreviation definition).

It would seem sensible to allow where clauses on both predicates and expressions, but to restrict the form of the local definitions to abbreviation definitions (or something equivalent). The obvious ways of extending the abstract syntax to handle this give no problems for concrete syntax, except that of deciding whether to use *let* or *where* (i.e. whether to put the local definitions before or after the expression or predicate). The choice is to some extent. a matter of taste. However, with *where* there is no nice way of finding out where the expression or predicate ends, whereas this is easy with *let*. *let* also gives a definition-before-use style, but, despite these advantages, the *where* form has been traditional in the Z community.

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
LocDef.1	No local definitions	*1*	*1*	*2*	*2*
LocDef.2	Provide *where*-expressions and *where*-predicates	*1*	*2*	*2*	*1*
LocDef.3	Provide *let*-expressions and *let*-predicates	*2*	*2*	*2*	*2*

In all of the above proposals, I take it that the local definitions comprise a list of abbreviation definitions.

6 Projection and Construction Operations

Z provides labelled and unlabelled n-ary products (i.e. schema types and tuple types). One might naturally expect there to be projection and construction operations for both sorts of product. Rather asymmetrically, Z provides projection operations for schema types and construction operations for tuple types but does not supply the other two combinations. It would seem good to make good these two omissions.

The projection operation for tuple types is very naturally given by extending the notation $b.c$ so that b is allowed to be a tuple and c is allowed to be a numeric constant. 1-based indexing should be used (as for sequencing) so that, e.g., $(a, b, c).\mathit{2} = b$.

The construction operation for schema types is a more serious omission. Without a notation for writing down a binding, it is hard to explain to a beginner what a binding is or to explain how the θ-construct works or indeed of what schemas are. Moreover, it is difficult to exhibit an element of a schema (e.g. in an implementability proof) or to write down the result of modifying a single component of a binding. Since both bindings and tuples are variants of the product idea, it would seem appropriate to base the syntax for binding construction on that for tuples, thus:

Example

$$(a \mathrel{\widehat=} \mathit{12},\ b \mathrel{\widehat=} \langle \mathit{1},\ \mathit{2},\ \mathit{3} \rangle,\ c \mathrel{\widehat=} \{\mathit{1}\})$$

denotes an element of the schema $[a : \mathbb{Z};\ b : \mathit{seq}\ \mathbb{Z};\ c : \mathbb{P}\mathbb{Z}]$. Here I am assuming adoption of proposal *SchPar.2* from section 10.3 below, if that proposal is not adopted then the example should read:

Example

$$(a == \mathit{12},\ b == \langle \mathit{1},\ \mathit{2},\ \mathit{3} \rangle,\ c == \{\mathit{1}\})$$

My view on the relative merits of these proposals is as follows:

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
PrjTpl.**1**	No projection operations for tuples	*1*	*0*	*0*	*0*
PrjTpl.**2**	Provide projection operations for tuples	*1*	*1*	*0*	*0*
ConBdg.**1**	No construction operation for bindings	$-\mathit{2}$	$-\mathit{2}$	*0*	*0*
ConBdg.**2**	Provide construction operation for bindings	*2*	*2*	*0*	*0*

The main reason for *PrjTpl.2* is symmetry, but that's quite an important factor in language design.

6.1 Bindings

It has been proposed by S. Brien that the notation for selecting a component of a schema be extended to allow an arbitrary expression to be used after the '.'. Under this proposal, if b is a binding and c is any term then $b.c$ denotes a term in which b is essentially treated as a local definition giving an environment in which to type check and evaluate c. For example, using the notation of section 6,

Example

$$(a \mathrel{\widehat=} \mathit{12},\ b \mathrel{\widehat=} \langle \mathit{1},\ \mathit{2},\ \mathit{3} \rangle,\ c \mathrel{\widehat=} \{\mathit{1}\}).(c \cup \{a\}) = \{\mathit{1},\ \mathit{12}\}$$

The expressive power gained by this extension is quite considerable. However the syntax is perhaps rather opaque, except to the specialist. Moreover, one loses the useful check on an expression like $S.comp$ that $comp$ is actually one of the signature variables of S. Since the new construct is actually a form of local definition, a "heavier" syntax might be better for ordinary mortals. One possibility is as shown in the following example:

Example

$$\mathit{open}\ (a \mathrel{\widehat=} \mathit{12},\ b \mathrel{\widehat=} \langle \mathit{1},\ \mathit{2},\ \mathit{3} \rangle,\ c \mathrel{\widehat=} \{\mathit{1}\}) \bullet (c \cup \{\mathit{2}\})$$

Here the keyword *open* is intended to be suggestive of "opening up" the binding to bring its signature variable into scope, and the overall format is intended to be reminiscent of a *let*-expression in a functional language and of the *with*-statements in Pascal or Ada.

It is perhaps helpful for pedagogical reasons that the *where* or *let* statement is effectively a special case of *open*.

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
Bndng.**1**	Allow bindings as local definitions, extending selection notation	*1*	*1*	*0*	*0*
Bndng.**2**	Allow bindings as local definitions using *open*-notation	*2*	*2*	*0*	*0*

7 PREDICATES

7.1 Association Rules for Logical Connectives

See 4.2.

7.2 Schema Expressions as Predicates

Like schema expressions as expressions these may sensibly be allowed in the abstract syntax, and if so, the concrete syntax must provided for them.

8 COMMENTS

Practical use of Z shows that it would occasionally be useful to have a facility for including commentary within a box. An example might be when one wishes to label the cases in a complex case analysis in order to ease discussion of the cases in the narrative part of the document.

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
Cmmnt.1	Support "inline" comments	*2*	*2*	*2*	*0*

9 SCHEMA EXPRESSIONS

There seem to be a couple of issues about the schema calculus as given in [4].

9.1 Schema Piping

The piping operation, which connects the outputs of one schema to the inputs of the next, is popular with many users, but is not supported by [4]. It would seem best to allow it, if for no other reason than that it is the only part of the language which gives any real support for the use of ? and ! as decoration for inputs and outputs.

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
SchPip.1	Schema piping operation supported	*2*	*2*	*0*	*0*

9.2 Expressions as Schema Expressions

There seems to be no reason for not allowing an arbitrary expression whose type is appropriate to be used as a schema expression. This does require the abstract syntax to be widened to allow it, but is semantically straightforward and conceptually beneficial (since it helps to demystify the notion of schema).

9.3 Δ and Ξ

The Δ and Ξ conventions of [4] give a form of implicit definition facility. This does not fit very smoothly into a rigorous definition of the language.

Three possibilities are suggested:

DelXi.1 Simply drop the convention: if the user wants to use ΔS or ΞS then they are short and easy to define.

DelXi.2 Adopt the convention: i.e., in the semantics for schema references say that ΔS and ΞS have the conventional meanings if they are not defined otherwise in the context.

DelXi.3 Integrate the convention into the language more closely by making Δ and Ξ new forms of schema expression.

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
DelXi.1	No Δ and Ξ convention.	*2*	*1*	$-$*1*	*2*
DelXi.2	Δ and Ξ convention as semantic special cases.	$-$*1*	*1*	$-$	$-$*1*
DelXi.3	Δ and Ξ as forms schema of schema expression.	*2*	*1*	*1*	*0*

10 PARAGRAPHS

10.1 Free Types

The chevron symbols in free type definitions serve little purpose. Thus:

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
FreeType.1	Free type definitions as per [4]	*1*	*1*	*0*	*0*
FreeType.2	Free type definitions as per [4] but not requiring chevrons.	*2*	*2*	*0*	*0*

FreeType.2 works particularly well in combination with *Fixity.3*, provided *FreeType.2* is formulated so that the right hand of a free type definition is just a list of expressions separated by | characters, with each expression expected to look like an application of the constructor it defines to an argument which actually give the type of the argument. An example might be:

$$\begin{array}{|lll}
[\mathit{Var}] & & \\
\mathit{fun} & \mathit{10} & \_ +_e \_ \\
\mathit{fun} & \mathit{20} & \_ *_e \_ \\
 & & \\
\mathit{Exp} & ::= & \mathit{Var} \\
 & | & \mathit{Exp} +_e \mathit{Exp} \\
 & | & \mathit{Exp} *_e \mathit{Exp}
\end{array}$$

Since free type definitions are not being used where they might be in the Z standard currently because of the verbosity of the syntax in [4], a more natural syntax for them does seem worthwhile.

10.2 Predicate Stacking

[4] allows line breaks or ';'s to be used to act as separators in the axiom part of an axiomatic box and similar. This effectively makes a line break or ';'s act as a symbol for conjunction which has lower precedence than the quantifiers, and is quite useful. Since I consider that Z should be free format as discussed in 2.2 above, there are two options:

PredSep.1 to insist on a ';' between the predicates in an axiom part.

PredSep.2 to allow the ';' between two predicates in an axiom part to be replaced by white space if it is clear from the context where the ';' must go.

PredSep.2 is not easy for the implementer of tools, nor for the reader of specifications which exploit it:

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
PredSep.1	';' between predicates necessary	*2*	*2*	*−1*	*2*
PredSep.2	';' between predicates optional.	*1*	*2*	*0*	*−2*

10.3 Paragraph Forms

In [1], abbreviation definitions (there called syntactic definitions) have essentially the same syntax as horizontal schema definitions. In [4], these two forms of definitions are syntactically different. Since both forms are semantically the same the distinction seems to be pointless, and so the two should surely be merged as in [1].

Another distinction made in both [1] and [4] is between generic boxes and axiomatic boxes. The distinction is presumably there because because generic definitions are, in both treatments, not allowed to be loose. This stipulation arises, I believe, from the assertion in [3] that justifying the consistency of a loose generic definition requires the axiom of choice. While choice might, perhaps, be required for a general metatheoretic result in this area, there does not seem to be any difficulty in demonstrating consistency of a range of useful loose generic definitions, and so on the grounds that over-specification is a bad thing, it would seem sensible to allow loose generics. An example of a useful looose generic definition might be:

$[X]$

$$pick : seq_1\ X \rightarrow X$$

$$\forall s : seq_1\ X \bullet pick\ s \in ran\ s$$

There is no problem in demonstrating the consistency of *pick* using a uniquely specified generic function of the same type as a witness.

By allowing loose generics, axiomatic boxes become a special case of generic boxes (viz., the ones with no formal parameters) and the form of box with parallel lines and no formal parameters becomes redundant.

Thus we have the following proposals:

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
SchPar.1	Distinguish schema and abbreviation definitions syntactically	*−1*	*−1*	–	*−1*
SchPar.2	Use same syntax for schema and abbreviation definitions	*2*	*2*	*−1*	*1*
GenPar.1	Distinguish generic and axiomatic box	*−2*	*−2*	–	*−1*
GenPar.2	Use same syntax for generic and axiomatic box	*2*	*2*	–	*1*

10.4 Grouping of Paragraphs

It is extremely desirable to allow mutual recursion between free type definitions, since dealing with the syntax of languages of various kinds is very important and many languages of interest naturally required several mutually recursive data types for their syntactic domains, Z itself being an example.

Moreover, some existing specifications have used mutual recursion for other sorts of paragraph. Some means of grouping several paragraphs together would seem to be useful (this may or may not need to be reflected in the abstract syntax, depending on the approach taken to define the context conditions).

Another, purely syntactic reason for having a means to group paragraphs is caused by the rules which allow generic actual parameters to be omitted. The rules in [4] mean that generic actual parameters may be omitted if their types are uniquely determined by the context, but leave open the question of how much context one needs to examine. Consider, for example, the following, in which one should note that g is *not* generic:

$$\mathit{Perm}[X] \mathrel{\widehat{=}} X \rightarrowtail X$$

$$g : \mathit{Perm}$$

$$g^{-1} \circ g = \mathit{Id}$$

$$h : \mathit{Perm}[\mathbb{Z}]$$

$$h = g^{-1}$$

Here there is only one possible type for the omitted generic actual parameter of *Perm* in the box for g, but that is not the case if we consider the first two paragraphs on their own. Since it is unreasonable to expect either the reader or a type checker to have to scan the entire specification for this sort of purpose, it would seem useful if paragraphs intended for treatment as a unit for type checking purposes could be grouped together in some way.

The only reasonable alternative to having a notation for grouping from the type-checking point of view seems to be to consider each paragraph to be in a group of its own for type checking purposes, and this is perfectly viable.

A third reason for allowing paragraphs to be grouped is that some users may wish to indicate, e.g., to a proof tool, what the 'units of conservative extension' in their specification are intended to be.

As pleasant a way of any of indicating grouping of paragraphs is to use '&' characters between paragraphs which are to be taken as a group. An example of mutually recursive free type definitions using this notation might be.

$$\begin{array}{llll}
 & \mathit{Cmd} & ::= & \mathit{skip} \\
 & & | & \mathit{assign}\ \langle\langle \mathit{Var} \times \mathit{Exp} \rangle\rangle \\
 & & | & \mathit{block}\ \langle\langle \mathit{Blk} \rangle\rangle \\
\& & \mathit{Blk} & ::= & \mathit{let}\ \langle\langle \mathit{seq}(\mathit{Var} \times \mathit{Typ}) \times \mathit{seq}\ \mathit{Cmd} \rangle\rangle
\end{array}$$

An example of the use of the notation to indicate a unit of conservative extension might be:

$$[\mathit{Inf}] \quad \&$$

$$\exists f : (\mathit{Inf} \rightarrowtail \mathit{Inf}) \setminus (\mathit{Inf} \twoheadrightarrow \mathit{Inf}) \bullet \mathit{true}$$

Here we wish to say that the given set *Inf* is infinite. We do it by first defining *Inf* and then asserting the existence f to be a one-one endofunction of *Inf* which is not onto. Now, the second paragraph constrains *Inf* and so is not a conservative extension of the first. However, the two paragraphs taken together are a conservative extension of the mathematical toolkit ($\mathbb{N}$ and $\mathbb{N} \lhd succ$ supply suitable witnesses for *Inf* and f). By grouping the paragraphs we can hint to a a proof tool that we want a consistency proof obligation to be generated for the two paragraphs taken together.

It may be felt that the ability to group together arbitrary lists of paragraphs is a little too controversial. Sensible compromises are to permit mutually recursive free types and/or to permit a given set declaration to be grouped with an immediately following constraint.

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
ParGrp.1	No facility to group paragraphs for type checking etc.	*0*	*−2*	−	*0*
ParGrp.2	Paragraphs may be grouped for the purposes of type checking etc. using '&'	*2*	*2*	*2*	*−1*
ParGrp.3	Mutually recursive free type definitions supported using '&'	*2*	*1*	*2*	*−1*
ParGrp.4	Given set paragraph may be grouped with constraint using '&'	*2*	*1*	*2*	*−1*
ParGrp.5	*ParGrp.3* and *ParGrp.4*.	*2*	*2*	*2*	*−1*

10.5 Theorems and Conjectures

[1] and [2] both give a syntax for including theorems in a specification. Since it is not very clear what the status of such a theorem should be, it is, perhaps, more appropriate to talk in terms of conjectures rather than theorems. Conjectures also allow more flexibility in informal discussion (as in "..., however, the following conjecture turns out false", "... and so we have proved, informally, the conjecture *lemma22*"). The notation for conjectures or theorems, should allow them to be generic in order that properties of generic constants may be expressed.

The significance of a conjecture would be informal (as is that of theorems in [1, 2], but allows the user to have conjectures type-checked. A proof tool could use the conjectures in a specification as a point of departure for proof work.

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
ThmCnj.1	Allow theorems as a paragraph form	*1*	*1*	*0*	*0*
ThmCnj.2	Allow conjectures as a paragraph form.	*2*	*2*	*0*	*0*

10.6 Constraints

[4, 1] do not support generic constraints. It would seem natural to give a notation for these.

Name	Description	*UND*	*EXP*	*CNV*	*IMP*
Cnstr.1	Allow generic constraints	*1*	*1*	*1*	*1*

A A PROPOSED CONCRETE SYNTAX FOR Z

In this appendix, I present a complete proposal for a concrete syntax for Z. The syntax is intended to give a reasonably rational set of decisions on the issues raised earlier in this document.

The syntax is presented in a style which is somewhat different from [4] or the draft standard [7], both of which tend to use BNF to describe precedence rules. This, in my opinion, is rather prone to obscure what's going on when used to excess. The style adopted here is to use an ambiguous grammar and to give disambiguation rules where necessary — clearly such a style is required if one is to handle user-defined operator precedence, as is the intention here. I have also tried to make the grammar compact by using optional and repeated constructs "inline" rather than having separate productions for pieces of a construct, wherever convenient.

I would like to stress that, while the syntax presented here may look a little unfamiliar, the main aim of the present proposal is to provide a coherent synthesis of the more successful features of earlier treatments. Some guiding principles in designing the syntax have been the following:

1. It should be possible to parse a specification without knowledge of the types of its constituents;
2. The language should be closed under the substitution of equals for equals;
3. The lexical rules should be simple, while offering support for a useful range of mathematical notations;
4. Compatibility with earlier versions of Z is desirable wherever it can be reconciled with the first three principles;
5. Where compatibility cannot be achieved, the incompatibilities should be such as can be readily described and detected.

[Explanatory remarks, like this one, which are to assist in evaluating the proposal, and which would be inappropriate for inclusion in the standard are given enclosed in brackets and in a small typeface. Many of these remarks use the mnemonic names for various proposals discussed in the main part of the document. An index to these mnemonics may be found in appendix A.4.15.]

A.1 Introduction

The concrete representation for Z is specified as follows:

1. Section A.2 describes the character set required to represent a Z specification.
2. Section A.3 describes the rules according to which character sequences are grouped into tokens.
3. Section A.4 gives a context-free grammar and context conditions which define which sequences of tokens may be viewed as Z specifications.

Application of the transformation rules to the parse tree obtained in stage 3 gives the abstract syntax representation of the Z specification.

The notation for the context-free grammar conforms to the BSI standard for grammars [5]. and may be summarised as follows:

- Sequences of specific terminal symbols are enclosed in single quotes, e.g., '*pre*'
- Terminal symbols standing for classes of symbols are represented by identifiers, as are nonterminal symbols, e.g., *Id*. Defining occurrences of these identifiers are shown in **bold** and an index to them may be found in appendix A.4.15
- Constructs to be concatenated are separated by a comma (',')
- Optional constructs are enclosed in square brackets ('[' and ']')
- Constructs which may be repeated (zero or more times) are enclosed in braces ('{' and '}')
- Alternative constructs are separated by a vertical line ('|')
- Subtraction of (the language generated by) one construct from another is denoted by '−'.
- Alternation has lower precedence than concatenation which has lower precedence than subtraction, but round brackets ('(' and ')') may be used to override this
- White space may be freely used, except within terminal and nonterminal symbols
- Comments are enclosed in '(∗' and '∗)'.

A.2 Character Set

At the most primitive level, a physical object (e.g, a document on paper or stored electronically) is interpreted as a finite sequence of *characters*. The method of deriving a sequence of characters from a physical object is not defined in this standard, however this section places minimum requirements on the character set.

The character set must include, at least, the characters in the sets *Letter*, *Greek*, *Digit*, *Symbol*, *Stop*, *Stroke*, *Subscript*, *Box*, *Quote*, *Ascii* and *Format* described in the following table.

Letter	A	B	C	D	E	F	G	H	I	J	K	L
	M	N	O	P	Q	R	S	T	U	V	W	X
	Y	Z										
	a	b	c	d	e	f	g	h	i	j	k	l
	m	n	o	p	q	r	s	t	u	v	w	x
	y	z										
Greek	α	β	γ	δ	ϵ	ζ	η	θ	ι	κ	λ	μ
	ν	ξ		π	ρ	σ	τ	υ	ϕ	χ	ψ	ω
			Γ	Δ				Θ			Λ	
		Ξ		Π		Σ		Υ	Φ		Ψ	Ω
Digit	0	1	2	3	4	5	6	7	8	9		
Symbol	↗	↕	∪	∩	⋃	⋂	⊆	⊂	𝔽	∅	∉	⦇
	⦈	⩤	⩥	◁	▷	⊢	⨾	⊕	⇸	→	⤔	↣
	⤀	↠	⤖	↔	ℕ	ℤ	≤	≥	<	>	⁀	⁀/
	≠	+	−	*	#	.	∼	⊎	⟦	⟧		
Stop	,	;	:	(	)	[	]	{	}	⟨	⟩	/
	\	_	∧	∨	¬	⇒	⇔	=	∈	∀	∃	•
	ℙ	×	≙	&	⊢	⨾	↾	::=	?⊢			
Stroke	′	?	!									
Subscript	Subscripted forms of any of the above characters.											
Box	<u>AX</u>	<u>SCH</u>	<u>END</u>	<u>IS</u>	<u>ST</u>	<u>BAR</u>						
Quote	"	'	(*	*)								
Ascii	A member of the ISO character set with code in the range 32 to 126.											
Format	A format character such as space, tab, line-break or page-break.											

The set *Ascii* may overlap other categories. This does not lead to any ambiguity because of the restricted use of *Ascii* in character and string literals.

[↗ and ↕ are characters to shift in and out of superscription. Transitive closure, reflexive-transitive closure and relational inverse can be written as ↗+↕, ↗*↕ and ↗∼↕, each of which is an identifier.]

[*Letter* might also include other fonts, e.g. italic or bold. If so, there is a question as to whether the standard should insist that, e.g., '**A**' be treated the same as 'A'?]

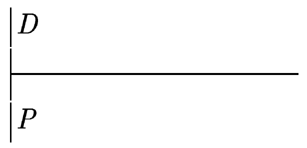
[The Greek letter omicron is not mandatory since it looks like an 'o' in some fonts.]

[The list of *Symbols* above should be extended in the actual standard to cover the requirements of the toolkit.]

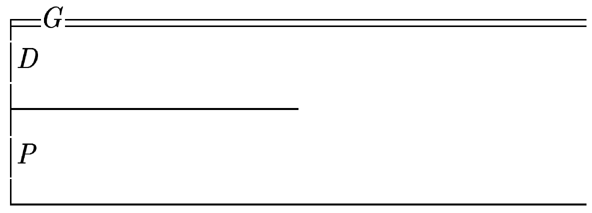
[<u>AX</u>, <u>ST</u> etc. are intended to represent characters for drawing boxes of various sorts.]

The following graphical conventions are adopted in this standard:

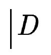
- Characters in the set *Subscript* are written in the subscript position.
- The characters ↗ and ↕ delimit sequences of characters to be written in the superscript position.
- If G, D, P and S arbitrary sequences of characters not containing any of the box characters (AX, BAR, ST, END, SCH and IS), then:
  - AX D ST P END is written as:



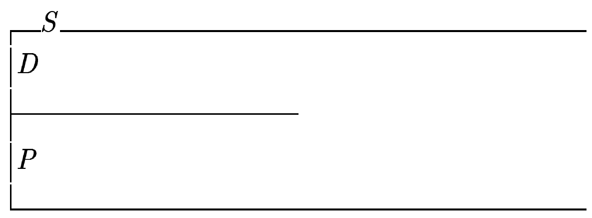
  - AX G BAR D ST P END is written as:



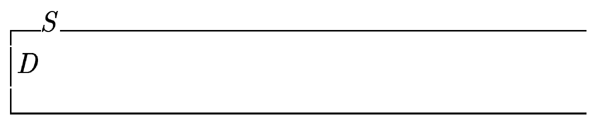
  - AX D END is written as:



  - SCH S IS D ST P END is written as:



  - and SCH S IS D END is written as:



[The above does not give it, but by adding an additional keyword to separate out the generic formals of an axiomatic box, the traditional form of a generic box could be adopted in the case where there are some generic parameters.]

A.3 Lexical Analysis

Token A *token* is a sequence of characters, as defined in section A.2, conforming to the grammar given in this section, whose terminal symbols are the sets of characters defined in section A.2, and whose sentence symbol is *Token*. The different sorts of token correspond to the sorts of terminal symbol listed in section A.4, together with an extra sort of space tokens.

A sequence of characters is interpreted as a sequence of non-space tokens by a left-to-right scan taking tokens which are as long as possible and then discarding any *Space* tokens. If it is not possible to do this then the sequence of characters is erroneous.

BNF

```
Token        =    Identifier
             |    Decor
             |    Narrative
             |    Number
             |    Character
             |    String
             |    Narrative
             |    Punctuation
             |    Space;
```

Identifier There are three sorts of identifier:

BNF

```
Identifier   =    Alphanumeric
             |    Greek
             |    Symbolic;

Alphanumeric   =    (Letter, {Letter | Digit | (‘_‘, (Letter | Digit)}), {Subscript};

Greek        =    Greek, {subscript};

Symbolic     =    Symbol, {Symbol}, {Subscript}
             |    Punctuation, Subscript, {Subscript};
```

[To maximise the flexibility of the language, particularly when used for the metatheory of itself or of other languages even a punctuation character can be used to form a symbolic identifier by attaching a subscript.]

[Since the mandatory Greek characters are insufficient for actually typing real Greek words (there being no breathings etc.), the view is taken that Greek letters work as in ordinary mathematics, $\alpha\beta\gamma$ containing three names. This seems to be a good compromise, and works nicely with λ, μ, identifiers.]

Decoration Decoration comprises just a sequence of stroke characters

BNF

```
Decor        =    Stroke, {Stroke};
```

[We are assuming that proposal *Decor.2* is adopted and that it is "implemented" in the transformation into abstract syntax. *Decor.3* is equally simple, and essentially just says that decoration is allowed at the end of an identifier as part of the identifier.]

Numbers A numeric literal is a non-empty sequence of decimal digits:

BNF

```
Number      =   Digit, {Digit};
```

Character A character literal denotes a Z character.

[To reason about numeric literals an axiom schema is required to act as the definitions of the numbers. How this is done is outside the scope of the present proposal.]

BNF

```
Character   =   ‘'‘, Char, ‘'‘;
```

A string literal denotes a sequence of Z characters:

BNF

```
String      =   ‘"‘, {Char}, ‘"‘;
```

BNF

```
Char        =   (     Letter | Greek | Digit | Symbol | Stop |
                      Stroke | Subscript | Quote | Ascii )
                 − (‘'‘ | ‘"‘ | ‘\‘)
            |   ‘\‘, ‘'‘
            |   ‘\‘, ‘"‘
            |   ‘\‘, ‘\‘;
```

[The intention here is that axioms and axiom schemata be available asserting that the distinct characters are all distinct and that the strings denote the free monoid over the set of characters in the obvious way. How this is to be done, and whether a more machine oriented axiomatisation relating characters to 8-bit bytes should be provided, is outside the scope of the present proposal.]

Narrative The means for delimiting the narrative sections between formal material in a Z document is not defined in this standard:

BNF

```
Narrative   =   ? Implementation Dependent ?
```

Punctuation This kind of token includes the stop and box characters of section A.2 symbols.

BNF

```
Punctuation =   Stop
            |   Box;
```

Space A space token is either a white space character or a comment. Comments may be nested.

BNF

```
    Space       =    Format | Comment;

    Comment     =    ‘(*‘, {Text}, ‘*)‘;

    Text        =    Letter | Greek | Digit | Symbol | Stop | Stroke
                |    Subscript | Quote | Ascii
                |    Quotation − (‘(*‘|‘*)‘)
                |    Format;
```

[*Comment* is for “inline” comments, as per proposal *Cmmnt.1*.]

A.4 Grammar

The following grammar defines a language over the set of all non-comment tokens as defined in section A.3. The specific terminal symbols of the grammar are the punctuation symbols and reserved identifiers listed below.

Punctuation

‘$\Leftrightarrow$‘	‘$\wedge$‘	‘$\vee$‘	‘$\neg$‘	‘$\Rightarrow$‘	‘$\forall$‘	‘$\exists$‘	‘$\mid$‘	‘$\bullet$‘	‘$\mathbb{P}$‘
‘$\times$‘	‘&‘	‘,‘	‘:‘	‘;‘	‘/‘	‘\‘	‘(‘	‘)‘	‘[‘
‘]‘	‘{‘	‘}‘	‘$\langle$‘	‘$\rangle$‘	‘_‘	‘$\in$‘	‘=‘	‘$\widehat{=}$‘	‘$\fatsemi$‘
‘$\upharpoonright$‘	‘::=‘	‘?$\vdash$‘	‘AX‘	‘SCH‘	‘END‘	‘IS‘	‘ST‘		

Reserved Identifiers

‘$\exists_1$‘	‘θ‘	‘λ‘	‘μ‘	‘Δ‘	‘Ξ‘	‘.‘	‘...‘	‘>>‘	‘*fun*‘
‘*gen*‘	‘*let*‘	‘*in*‘	‘*open*‘	‘*pre*‘	‘*rel*‘	‘*true*‘	‘*false*‘		

[The only difference between punctuation symbols and reserved identifiers is that the former are not identifiers according to the lexical rules given in section A.3 whereas the latter are. The grammar effectively prohibits attempts to use the reserved identifiers as variables.]

[*Ovrld.1* is implicit in the use of ‘$\upharpoonright$’ and ‘$\fatsemi$’ as punctuation.]

The general terminal symbols of the grammar are as shown in the following table.

Symbol	Description
Id	Identifier other than the reserved identifier
Decor	A sequence of decoration characters
Narrative	Narrative text.
Number	Numeric literal
Character	Character literal
String	String literal
Narrative	Informal text

A.4.1 Specification

A specification comprises a sequence of *paragraphs* interleaved with narrative text

BNF

Specification = [*Narrative*], {*Paragraph*, *Narrative*}, [*Paragraph*];

A.4.2 Paragraphs

A paragraph takes one of 7 forms:

BNF

```
Paragraph   =   Fixity
            |   GivenSet
            |   AbbDef
            |   FreeTypeDef
            |   AxBox
            |   Constraint
            |   Conjecture;
```

[Since the fixity paragraphs influence the forms of the declarations appearing in most other paragraphs they are listed and discussed first.]

A.4.3 Fixity Paragraph

A fixity paragraph describes syntactic abbreviations which are to be used in the specification. The constructs which can be abbreviated are application of a function to a tuple of arguments, explicit instantation of a generic constant and the membership predicate. These three possibilities are given by the *fun*, *gen* and *rel* options in the following, in which the *Number* gives a numeric precedence for the abbreviations being described. Omitting the *Number* is equivalent to supplying it as *0*.

BNF

```
Fixity      =   'fun' , [Number], Template, {',', Template}
            |   'gen', [Number], GTemplate, {',', Template}
            |   'rel', Template, {',', Template};
```

A template has the form of a sample use of the abbreviation with placeholders for the arguments. The placeholders are either '_', corresponding to an argument position where a single expression is expected, or '...', corresponding to an argument position requiring a list of expressions (a possibility which does not arise in *gen* fixity paragraphs).

BNF

```
Template    =   (['_'], {Id, ('_' | '...')}, Id, ['_']) − Id;

GTemplate   =   (['_'], {Id, '_'}, Id, ['_']) − Id;
```

The syntactic abbreviations introduced by a fixity paragraph are in force throughout the entire specification containing them. The following rules apply to the identifiers which appear in a template:

1. the first and last identifiers in the template must not appear anywhere in any other template in any fixity paragraph in the specification, unless that template introduces exactly the same syntactic abbreviation.

2. identifiers other than the first and last in the template must not appear as the first, or last, identifier in any template in the specification.

[This is proposal *Fixity.4* in action. I stress that this mechanism is just a generalisation of the familiar notation for infix, prefix and postfix operators covering the requirements of the mathematical toolkit and allowing clear expresion of which of the three sorts of operator is being introduced. The rules above are intended to enhance readability by not allowing the fixity of an identifier to vary.]

A.4.4 Given Set Definition

A given set definition lists the names of the given sets being defined, optionally followed by an axiomatic box or a constraint:

BNF

```
GivenSet    =    '[', Name, {',', Name}, ']', ['&', Constraint];
```

[The optional extra paragraph is for proposal *ParGrp.5*.]

A.4.5 Abbreviation Definition

BNF

```
AbbDef      =    EqDef
            |    SchemaBox;

EqDef       =    DefLhs, '≙', Expression;

SchemaBox   =    SCH, DefLhs, IS, Decl, [ST, AxiomPart], END;

DefLhs      =    ([Name], {Id, Name}, IdDec, [Name]) − Id
            |    Name, [GenFormals];

GenFormals  =    '[', Name, {',', Name}, ']';

AxiomPart   =    Pred
            |    Pred, ';', AxiomPart
```

An instance of the first alternative for *DefLhs* must match some template in some *gen* fixity paragraph in the specification, in the sense that the template results if we delete any decoration from the *DefLhs* and replace each *Name* in it by '_'.

[*SchPar.2* is implicit here in the absence of a form of *EqDef* using '==' rather than '≙'.]

[Here we have *PredSep.1*: a ';', but not a line break, may be used in a top level predicate as a symbol for conjunction (with very low precedence).]

A.4.6 FreeType Definition

BNF

```
FreeTypeDef=    Name, '::=', Branch, {, '|', Branch}, ['&', FreeTypeDef];

Branch      =   ([Expr], {Id, FreeTypePar}, IdDec, [Expr])
            |   Name, [Expr];

FreeTypePar=    Expr
            |   Expr, '...';
```

We say that the first alternative for a branch matches a template if the template may be obtained from the branch by deleting any decoration, replacing the optional expressions and the *FreeTypePar*s of the form *Expr* with '_', and replacing the *FreeTypePar*s of the form *Expr* '...' by '...'. Each branch construed under the first alternative must match some template in some *fun* fixity paragraph in the specification.

[Here we have *ParGrp.5* and *FreeType.2*.]

A.4.7 Axiomatic Box

BNF

```
AxBox       =   AX, [GenFormals, BAR], Decl, [ST, AxiomPart], END;
```

[This is *SchPar.2*, allowing axiomatic boxes to have generic parameters rather than thinking of generic boxes as a different category.]

A.4.8 Constraint

BNF

```
Constraint  =   [GenFormals], Pred;
```

[Here we have *Cnstr.1*: the constraint may be preceded by a list of generic formals.]

A.4.9 Conjecture

A conjecture comprises an optional label, an optional list of formal parameters and a predicate. A conjecture is purely for documentary purposes and has no effect on the abstract syntax form of the specification.

BNF

```
Conjecture  =   [Id], [GenFormals], '?⊢', Pred;
```

[This is *ThmCnj.2*]

[One should say, somehow, that the conjecture must be well-typed.]

A.4.10 Declaration

BNF

```
Decl        =   BasicDecl, {‘;‘, BasicDecl};

BasicDecl   =   Name, {‘,‘, Name}, ‘:‘, Expr
            |   Schema;
```

[Here *DecSep.1* has been adopted.]

A.4.11 Schema

BNF

```
Schema      =   Schema2
            |   Quant, SchemaText, ‘•‘, Schema;

Quant       =   ‘∀‘ | ‘∃‘ | ‘∃₁‘;

Schema2     =   Schema3
            |   Schema2, SchInOp, Schema2
            |   Schema2;

SchInOp     =   ‘>>‘ | ‘↾‘ | ‘⨾‘ | ‘∧‘ | ‘∨‘ | ‘⇒‘ | ‘⇔‘;

Schema3     =   Schema4
            |   SchPreOp, Schema3;

SchPreOp    =   ‘¬‘ | ‘pre‘ | ‘Δ‘ | ‘Ξ‘;

Schema4     =   Expr
            |   Schema4, [Decor]
            |   Schema4, [RenameList]
            |   ‘[‘, SchemaText, ‘]‘                     (* Schema4.A *)
            |   Schema4, ‘\‘, ‘(‘, Name, {‘,‘, Name}, ‘)‘;

RenameList  =   ‘[‘, Name, ‘/‘, Name, {‘,‘ Name, ‘/‘, Name}, ‘]‘;
```

The grammar for *Schema2* is ambiguous. The ambiguities are resolved by taking the alternatives for *SchInOp* as listed in decreasing order of precedence.

[It is unclear from [4] what the relative precedence of the schema calculus connectives should be. The above grammar and rule are modelled on [1], in which I read hiding as being a postfix operator. However, the treatment of decoration here is quite different and I have made renaming a high precedence construct, as in the draft standard, since this is more uniform with the visually similar hiding construct.]

[I have adopted *SchPip.1* by including schema piping:‘>>’.]

A.4.12 Schema Text

BNF

```
SchemaText  =   Decl, [ '|', Pred ];
```

A.4.13 Predicate

BNF

```
Pred        =   Pred1;

Pred1       =   Pred2
            |   Quant, SchemaText, '•', Pred1
            |   'let', EqDef, {',', EqDef}, 'in', Pred1
            |   'open', EqDef, {',', EqDef}, 'in', Pred1;

Pred2       =   Pred3
            |   Pred3, LogInOp, Pred3;

LogInOp     =   '∧' | '∨' | '⇒' | '⇔';

Pred3       =   Pred4
            |   '¬', Pred3;

Pred4       =   Expr, Rel, Expr {Rel, Expr}                  (* Pred4.A *)
            |   ([Expr], {Id, Exprs}, IdDec, [Expr])
                − (Id | (Expr, IdDec, Expr))                 (* Pred4.B *)
            |   Schema
            |   'true'
            |   'false'
            |   '(', Pred, ')';

Rel         =   IdDec | '∈' | '=';
```

The grammar for *Pred* is ambiguous. The ambiguities are resolved by imposing the following rules:

1. The operators in the production for *LogInOp* are listed in decreasing order of precedence, and are right associative.

   [This is *Assoc.2*.]

2. In *Pred4.B* the result of replacing all the expressions by '_' must be a template appearing in a *rel* fixity paragraph somewhere in the specification.

3. In *Pred4.A* the *Id* in each *IdDec* must appear in a template of the form _*Id*_ in a *rel* fixity paragraph.

4. A construct which can be construed both as a *Pred* and a *Schema* should be construed as a *Schema*, wherever possible.

   [The ambiguity which this rule resolves in fact has no effect on the meaning of a specification, since “promotion” of schemas to predicates commutes with the logical operations.]

A.4.14 Expression

BNF

```
Expr        =   Expr0;

Expr0       =   Expr1
            |   ‘μ‘, SchemaText, ‘•‘, Expr
            |   ‘λ‘, SchemaText, ‘•‘, Expr
            |   ‘let‘, EqDef, {, ‘,‘, EqDef}, ‘in‘, Expr
            |   ‘open‘, Expr, ‘in‘, Expr;

Expr1       =   Expr2
            |   ([Expr1], {Id, Exprs}, IdDec, [Expr1]) − Id;      (∗ Expr1.A ∗)
            |   Expr1, ‘×‘, Expr1, {‘×‘, Expr1};                  (∗ Expr1.B ∗)

Expr2       =   Expr3
            |   ‘ℙ‘, Expr2;

Exprs       =   [Expr, {‘,‘, Expr}];

Expr3       =   Expr4
            |   Expr3, Expr4
            |   ‘θ‘, Expr4, [Decor];

Expr4       =   Name, [GenActuals]
            |   Literal
            |   ‘(‘, Expr, {‘,‘, Expr}, ‘)‘
            |   Schema
            |   ‘(‘, EqDef, {‘,‘ EqDef}, ‘)‘
            |   ‘⟨‘, [Expr, {‘,‘, Expr}], ‘⟩‘
            |   ‘{‘, [Expr, {‘,‘, Expr}], ‘}‘                      (∗ Expr4.A ∗)
            |   ‘{‘, SchemaText, [‘•‘, Expr], ‘}‘                  (∗ Expr4.B ∗)
            |   Expr3, ‘.‘, Name;
            |   Expr3, ‘.‘, Number;

GenActuals  = ‘[‘, Expr, {‘,‘, Expr}, ‘]‘;

Literal     =   Number | Character | String;
```

The grammar for expressions is ambiguous, the ambiguities are to be resolved using the following rules.

1. The alternative *Expr1.A* is only allowed when the resulting phrase matches a template in a

gen or *fun* fixity paragraph in the specification. Here the phrase is said to match a template if the template can be obtained from it by ignoring decoration, replacing each direct constituent *Expr* in the phrase by ‘_’ and replacing each direct constituent *Exprs* by ‘...’.

2. *Expr1.A* with a template from a *gen* fixity paragraph has lower precedence than *Expr1.B* which has lower precedence than *Expr1.A* with a template from a *fun* fixity paragraph.

3. The precedences in the fixity paragraphs give the precedence to apply for a phrase which can be interpreted in two ways using *Expr1.A* with templates from fixity paragraphs of the same sort.

4. If a phrase has two interpretations under *Expr1.A* with the same template, then the right-associative interpretation is to be taken.

5. A phrase of the form $\{S\}$ where S is a *Name* could be interpreted as a set display (*Expr4.A*) or as a set comprehension (*Expr4.B*). The set display is to be preferred.

6. A phrase of the form $V[S]$ where V is a *Name* could be interpreted as a generic instantiation (*Expr4.A*) or as application of V to a horizontal schema (*Schema4.A*). The generic instantiation is to be preferred.

7. A construct which can be construed both as a *Schema* and an *Exp* should be construed as an *Exp*, wherever possible.

   [The ambiguity which this rule resolves in fact has no effect on the meaning of a specification, since “promotion” of expressions to schemas has no effect on the semantics .]

8. A phrase beginning with *let* or *open* must extend as far to the right as possible, so that the interpretation as a predicate is to be preferred over that as an expression where ambiguity arises.

[The following proposals are adopted here: *Fixity.4*, *Assoc.2*, *UnNeg.1*, *SingSetDisp.2*, *LamMuScp.2*, *LocDef.3*, *PrjTpl.2*, *ConBdg.2* and *Bndng.2*.]

[It seems a little odd that ‘.’ is like an infix operator, yet it binds tighter than function application, but I have nonetheless followed [4] in this.]

A.4.15 Names

BNF

```
Name    =    IdDec
        |    ‘(‘, (([‘_‘], {Id, (‘_‘ | ‘...‘)}, IdDec, [‘_‘]) − IdDec), ‘)‘;
```

In the first alternative the *Id* must not appear in any fixity paragraph in the specification. In the second form the result of deleting any decoration must make the phrase between the brackets the same as some template in some fixity paragraph in the specification.

[This is *VrDcNm.1*]

[Note that decoration on a mixfix operator is applied to the last identifier in the name.]

BNF

```
IdDec   =    Id, [Decor];
```

B INDEX