Z: GRAMMAR AND CONCRETE AND ABSTRACT SYNTAXES

(Version 2.0)

by

Steve King
Ib Holm Sørensen
Jim Woodcock

Technical Monograph PRG-68

ISBN 0-902928-50-3

July 1988

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Copyright © 1988 IBM United Kingdom Laboratories Limited

Authors' address:

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Electronic mail: king@uk.ac.oxford.prg (JANET)

Preface

This monograph, which presents a grammar and an abstract syntax for the Z specification language, is produced as part of a joint project between IBM United Kingdom Laboratories Limited at Hursley, England and the Programming Research Group of Oxford University Computing Laboratory, into the application of formal software specification techniques to industrial problems. The work was supported by a research contract between IBM and Oxford University and is published by permission of the Company.

[Abrial 81] provided the starting point in the development of the Z notation. The syntax for definitions, predicates and terms presented here was developed from Jean-Raymond Abrial's paper. The notation has been further developed and described in [Sufrin 86].

The type rules and the semantics of Z have been described in [Spivey 85]. The commentary in this paper on the meaning of the language constructs is an informal description of what is formally described in [Spivey 85].

The schema concept is an extension to conventional set theory and preliminary descriptions can be found in [Sufrin 81], [Sørensen 82] and [Morgan 84]. A tutorial introduction to the present state of the schema notation can be found in [Woodcock 88].

Version 2.0

It is the authors' expectation (and hope!) that this will be the 'final' version of this document, at least in its present form. It is presented as part of the PRG's (and IBM's) work towards the standardisation of Z. As such, it has two major aims: to capture the present state of the language, particularly those parts of the language whose syntax has become stable, and to suggest possible solutions to several problems which have to be resolved as part of the standardisation process. In this second category come such topics as the syntax for theorems (which is dependent, to some extent, on agreement on a logic for Z), mnemonic names for the many non-ASCII symbols used in Z, and the whole question of how to use one Z document within another (ie imports, document qualifiers, versions etc). This last item can only be resolved when case studies have been completed, using for instance a library of specifications. To repeat: in these cases, what is presented in this document is merely a suggested solution—the definitive answer can only appear in due time!!

Contents

1 **Introduction** 3

2 **Syntax** 5
- 2.1 A document . . . 5
- 2.2 Identifiers, names and references . . . 8
- 2.3 Definitions and declarations . . . 9
- 2.4 Theorems . . . 14
- 2.5 Predicates . . . 15
- 2.6 Terms . . . 18
- 2.7 Schema terms . . . 24

3 **Terminal symbols** 29
- 3.1 Document punctuation . . . 29
- 3.2 Identifier lists and identifier symbols . . . 29
- 3.3 Definitions and declarations . . . 30
- 3.4 Theorem symbols . . . 30
- 3.5 Predicate symbols . . . 31
- 3.6 Term symbols . . . 31
- 3.7 Schema notation . . . 31

4 **Symbols defined in the Basic Library** 33
- 4.1 Set notation . . . 33
- 4.2 Relation notation . . . 33
- 4.3 Function categories . . . 34
- 4.4 Natural numbers . . . 34
- 4.5 Sequence and bag notation . . . 35

5 **An Abstract Syntax for the Z Notation** 36
- 5.1 A document . . . 37
- 5.2 Identifiers, names and references . . . 38
- 5.3 Definitions and declarations . . . 39
- 5.4 Theorems . . . 40
- 5.5 Predicates . . . 41
- 5.6 Terms . . . 42
- 5.7 Schema terms . . . 43

Acknowledgments 44

References 44

Index of definitions 45

Summary of syntax 47

1 Introduction

In this monograph we give a comprehensive description of the syntax for the specification language Z. The language has already been described informally in [Sufrin 86] and in [Hayes 87].

The aim of this document is to prepare for the standardisation of the syntax of Z. The abstract syntax of Z is defined and a concrete representation of all language constructs is given. The paper suggests a terminology for the constructs and the concepts of the Z language. The syntax describes a language which is as close as possible to that which is used in existing case studies. However, some syntactic variants of constructs in the language which have been used in the past have been omitted.

The paper gives an informal introduction to the scope rules for identifiers, the type rules for terms and the precedence rules for operators, functions and relation symbols. This description is informal, and the description of the type rules is concise and complex, and may be passed over if the reader desires.

Chapter 2 defines the concrete syntax of the language.

2.1 defines the overall structure of a Z specification docnment and describes the general facilities in Z for introducing new concepts.

2.2 sets some conventions for the naming of variables in Z documents.

2.3 gives the syntax for introducing new constants, data types and schemas. It describes how Z can be extended to include new infix binary operator symbols, new prefix and postfix function symbols and new infix relation symbols. It also describes how to introduce generic sets, functions and relations.

2.4 describes the syntax used for theorems.

2.5 gives the syntax of predicates. The predicate notation is conventional, although the use of types and constraints in quantified expressions is peculiar to Z.

2.6 gives the syntax for terms which is the notation used to describe elements and sets. This is conventional, for the most part, and includes the notion of types.

2.7 describes the notation for schemas. The schema notation is the part of the Z language which is used to give names to reusable specifications written in the set theoretical notation. Specifications written in Z make extensive use of the schema notation.

Chapter 3 describes the terminal symbols of this syntax.

Chapter 4 contains a description of the symbols and identifiers which have been given special meaning in the basic library.

Chapter 5 presents an abstract syntax for Z; the structure of this chapter follows that of chapter 2.

2 Syntax

The notation used to describe the syntax of Z is a simple variant of BNF:

- non-terminals are words in lower case letters, possibly with embedded underscores.
- alternative constructs are denoted by | at the start of a line.
- optional constructs are enclosed in angle brackets ⟨ ⟩ .
- space denotes zero or more spaces or continuation characters ('soft' newlines).
- terminals: 1) special characters; 2) reserved words, shown in **bold**; and 3) UPPER CASE abbreviations, which stand for large graphical forms (such as 'top of box') or invisible symbols (such as newline). All terminal symbols are listed in chapter 3, section by section, i.e. symbols from section 2.1 are listed in 3.1, those from 2.2 are listed in 3.2 etc.

To help in cross-referencing, page references are given for each syntactic category on the right hand side of a production. These references may be found in the right hand margin.

2.1 A document

```
document      ::=   explan_text ⟨ document ⟩                        5 5
                |   Z z_text EZ ⟨ document ⟩                        5 5

explan_text   ::=   "sequence of characters, excluding Z and EZ"
```

A document is divided into a series of sections, which may be: 1) explanatory text (explan_text), which is a description in a natural language; or 2) Z text which gives a formal description in the language Z.

```
z_text        ::=   ⟨ z_phrase ⟩ ⟨ list_sep z_text ⟩               6 5 5

list_sep      ::=   ;
                |   NL
```

Z text is a sequence of Z phrases separated by semicolons or newlines[1].

[1] If a Z phrase is to be split between two or more lines, a 'soft' newline character is used as a continuation character.

```
z_phrase      ::=  given_set_def                                6
                |  definition                                   6
                |  constraint                                   7
                |  theorem                                      14
                |  import                                       7
```

A Z phrase can take the following forms:

1. Given Set Definition

   ```
   given_set_def  ::=   [ name_list ]                           8
   ```

   The given sets of a document appear as a list of base names enclosed in square brackets ([,]). The given sets are formal parameters to the document. When a document is imported into another document, a set can be supplied as the actual value for a given set.
   Scope: The scope of a given set is the entire document.
   Type[2]: A given set is a type. If T is a given set, then the elements of T have type T; T itself has type $\mathbb{P}$(T).

2. Definition

   ```
   definition     ::=   axiomatic_def                           9
                     |  syntactic_def                           10
                     |  datatype_def                            11
                     |  schema_def                              12
   ```

   This language category is used to define constants, data types and schemas. The syntax for defining them is described in section 2.3. Constants are introduced by axiomatic definitions and syntactic definitions and in data type definitions. Data types are introduced by data type definitions and schemas are introduced in schema definitions. Note that a data type definition introduces both a data type and some constants. Axiomatic, syntactic and schema definitions may be generic.
   Axiomatic definitions also allow us to extend the language with infix operator symbols (op), prefix and postfix function symbols (func, pfunc), and infix relation symbols (rel). When generically defined operator, function or relation symbols are used, they are implicitly instantiated and appear in infix, prefix or postfix form (e.g. $f \oplus g$, $\#S$, r^{-1}, $s \subseteq t$). Syntactic definitions also allow us to extend the language with new symbols: infix symbols for naming sets (in), e.g. $\twoheadrightarrow$; prefix symbols for naming sets (pre), e.g. $\mathbb{P}1$; postfix symbols for naming sets (post). When these symbols are used, they must be explicitly instantiated (e.g. $\mathbb{N} \twoheadrightarrow \mathbb{N}$ is

[2] Z is based on typed set theory, in which all elements are associated with a particular maximal set (their type) from a universe of sets. The type of any element is unique and can always be determined. There is no set in Z to which all elements belong, instead there is a universe of sets, called types, which consists of all given sets, data types and schema types, and all sets which can be constructed from them using $\mathbb{P}$ (power set), $\times$ (cartesian product) and bracketing (()).

the set of partial functions over $\mathbb{N}$, $\mathbb{P}X$ is the set of subsets of X).
Scope: The scope of a constant, data type or schema is the entire document. They are referred to as global variables. Note that only data types may be defined recursively. When a document is imported, the scope of these global variables is extended to the importing document. Scope rules follow the static structure of the document: new local mathematical variables may be introduced by declaration in constructs nested within the document (e.g. $\forall$, $\exists$). Such constructs are excluded from the scope of any global variable if its name is reused for a local variable (see also section 2.3.
Type: The types of constants, data types and schemas are described in section 2.3.

3. Global Constraint

```
constraint      ::=    pred1                                          15
```

Constants are assumed to satisfy all the global constraints of the document.

4. Theorems
A theorem is a formal statement about the definitions given in the document (see section 2.4).

5. Import

```
import          ::=    base_name ⟨ version ⟩ ⟨ instantiation ⟩        888
```

A document is imported by giving its name. The uninstantiated given sets, constants, data types and schemas of the imported document become globally known to the importing document. However, if a version is specified, then the version name is added to the imported identifiers, and the identifiers are known only in their renamed form. Each of the given sets of the imported document may be instantiated i.e. a value for each of the given sets may be supplied (in an instantiation, see section 2.2), thereby achieving a partial instantiation of the imported document.
Scope: The use of the import facility may lead to ambiguity as variables from different documents can have identical names. This is resolved by prefixing the reference to such a variable with the name of the document which introduces it (`references` are described in section 2.2).
Type: Instantiation of given sets is systematic, and may add more information about the types used in the imported document. For example, if an imported variable `x` is of type `P(S×T)` where `T` is a given set of the imported document, and if `T` is instantiated with $\{1, 2, 3\}$, then the type of `x` will be `P(S×N)`.

2.2 Identifiers, names and references

id_list	::=	id ⟨ , id_list ⟩	8 8
id	::=	base_name ⟨ decor ⟩	8 8
base_name	::=	"sequence of alphanumerics, underscores and symbols, excluding the terminal symbols of this syntax and those symbols which have been defined as operators"	
name_list	::=	base_name ⟨ , name_list ⟩	8 8

An identifier list (id_list) is a sequence of identifiers (id), separated by commas. An identifier consists of a base name, which may be decorated (decor). Identifiers are used to name mathematical variables (e.g. elements, sets). When naming schemas, only base names can be used, since decoration is interpreted as a decoration operation which renames the bindable variables of the schema (see section 2.7.2). Note that some identifiers (e.g. dom, seq) have been given a meaning in the basic library. Strings of digits will be interpreted as numbers in base 10. Numbers are described in the basic library.

decor	::=	version ⟨ decor ⟩	8 8
	|	attribute ⟨ decor ⟩	8 8
version	::=	V base_name EV	8
attribute	::=	!	
	|	?	
	|	′	

A base name decoration is a sequence of version or attribute decorations. A version decoration is a subscript name. An attribute may contain exclamation marks (for output variables), question marks (for input variables) or dashes (for variables denoting resulting states).

reference	::=	⟨ doc_qual $ ⟩ id ⟨ instantiation ⟩	8 8 8
doc_qual	::=	base_name ⟨ version ⟩	8 8
instantiation	::=	[inst_list]	9

```
inst_list     ::=  term_list                                        9
                |  binding_list                                     9

term_list     ::=  term ⟨ , term_list ⟩                             19 9

binding_list  ::=  id = term ⟨ , binding_list ⟩                     8 19 9
```

When a reference is made to a variable, its name may be prefixed with a document qualifier to indicate the document in which the variable is defined. Generically defined variables may be instantiated. The document qualifier is terminated by a dollar. The document qualifier consists of the name of the document together with an optional version decoration. Each element of an instantiation list (`inst_list`) gives values to the generic parameters of a generic definition.

```
schema_ref    ::=  ⟨ doc_qual $ ⟩ base_name ⟨ instantiation ⟩       8 8 8
```

A reference to a schema can not include any decoration, since decoration of schema names is interpreted as a special operation (see section 2.7.2).

2.3 Definitions and declarations

Definitions introduce constants, data types and schemas. Definitions of constants and schemas may be generic. A definition takes one of the following forms:

1. Axiomatic definition

   ```
   axiomatic_def ::=  liberal_def                                   9
                   |  generic_def                                   9

   liberal_def   ::=  dec ⟨ | pred ⟩                                13 15
                   |  SR v_sch_text ER                              12

   generic_def   ::=  decl_id params :  term ⟨ | pred ⟩             13 9 19 15
                   |  SR ⟨ params ⟩ GE v_sch_text ER                9 12

   params        ::=  [ name_list ]                                 8
   ```

   An axiomatic definition introduces constants in global declarations (`dec` and `dec_list`). Constants can be elements, sets or functions which are used in prefix (`func`), infix (`op`) or postfix (`pfunc`) forms. The constants satisfy all their defining predicates. An axiomatic definition takes one of two forms: a liberal def-

inition, which introduces constants and, for each constant, gives a set from which the value of the constant must be drawn; or a generic definition, which introduces a *family* of constants, parameterised by the generic parameters of the list params. Each instance of a generic constant has a *unique* value.
Scope: The scope of a constant is the entire document (global scope), excluding the declaration list in which it is declared and any construct in which its name is reused for a local variable. The parameters of a generic definition are local to the definition, but they can be given values when the generic constant is instantiated.
Type: The type of constants introduced in a declaration is described in section 2.3.2.
Appearance: When a definition is given in a 'vertical' form, it is enclosed by SR (Start vertical Rule) and ER (End vertical Rule). In the case of a generic definition, SR and ER also denote the scope of the generic parameters.

A liberal definition may appear in either a horizontal or a vertical form:

$x : \mathbb{N} \mid x \neq 1$

or

$x : \mathbb{N}$

$x \neq 1$

A generic definition may also appear in a horizontal or a vertical form:

$tail[X] : seq\ X \rightarrow seq\ X \mid predicate$

or

$[X]$
$_\cup_ : \mathbb{P}X \times \mathbb{P}X \rightarrow \mathbb{P}X$

---

predicate

2. Syntactic definition

```
syntactic_def ::=   decl_id ( params ) ≙ term          13 9 19
                  | const_sym ≙ term                    11 19
```

A syntactic definition defines a new constant, which can be generic. It may also define a generic constant which has been given a symbol name.

Scope: A constant which is defined by a syntactic definition has global scope, excluding constructs in which its name is reused for a local variable. Note, also, that recursive syntactic definitions are not allowed.
Type: The constants defined in a syntactic definition have the same type as their defining term. Types of terms are described in section 2.6.

`const_sym`	`::=`	`pre id`	8
	`|`	`id post`	8
	`|`	`id in id`	8 8

A syntactic definition can be used to define a (possibly generic) constant which is to be named by an identifier (`decl_id`). Alternatively, it can be used to define a generic constant which will be named by a prefix, postfix or infix symbol (from the syntactic categories `pre`, `post` and `in`), in which case the names of the generic parameters (`id`) indicate the positions of the actual parameters which must be supplied when the set is used. For example, the definition

$$\mathbb{F}X \triangleq \{s : \mathbb{P}X \mid ...\}$$

introduces the (generic) set of finite subsets of a set. $\mathbb{F}A$ and $\mathbb{F}B$ are two different instantiations of this set. Chapter 4 gives a list of these special symbols which are introduced in the Z Basic Library.
Appearance: A syntactic definition has a horizontal form for constants named by identifiers: e.g.

$$some[X] \triangleq \{x : X \mid ...\}$$

and a horizontal form for constants named by symbols: e.g.

$$X \leftrightarrow Y \triangleq \mathbb{P}(X \times Y)$$
$$X \nrightarrow Y \triangleq \{X \leftrightarrow Y \mid ...\}$$

$\mathbb{N} \nrightarrow \mathbb{N}$ is an instantiation of $\nrightarrow$ and is the set of all partial functions from $\mathbb{N}$ to $\mathbb{N}$.

3. Data type definition

`datatype_def`	`::=`	`id ::= branches`	8 11
`branches`	`::=`	`id ⟨ ≪ term ≫ ⟩ ⟨ | branches ⟩`	8 19 11

A data type definition introduces a new set, known as a data type. It is formed from the union of constants and the ranges of new constructor functions. In the definition, each new constant appears as an identifier, while each constructor function appears in a branch with a description of its domain enclosed in double angled brackets (≪ ... ≫).
Scope: The data type itself, as well as all the new constants and constructor func-

tions, has global scope. Recursive data type definitions are allowed.
Type: A data type is a type. The following example explains the types of the constants introduced by a data type definition. Suppose

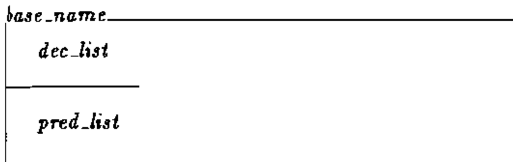
```
DT ::= a | b | f ≪N≫
```

This definition introduces the constants a and b of type DT and the function f with domain N. f is an element of N ↣ DT and so has type P(N×DT).

4. Schema definition

schema_def	::=	base_name ⟨ params ⟩ ≙ schema_term	8 9 24
	|	base_name ⟨ params ⟩ SB v_sch_text ESB	8 9 12

A schema definition introduces either a single new schema or a family of schemas, and names them. A schema name can be used as a predicate, as an inclusion in a declaration list, as a term in a θ-construction or as a term in a declaration.
Scope: The schema is globally known, but recursive schema definitions are not allowed. The variables introduced in the declaration list of the schema text (dec_list) are local to the following predicate list (pred_list).
Type: When a schema is used as a term in a declaration, it denotes a type. This is described in sections 2.6 and 2.7.1. The type of a θ-construction is described in section 2.6.2.
Appearance: Use of SB, ST and ESB corresponds to



2.3.1 Schema texts

When variables are introduced in a declaration list, it is often necessary to give a predicate which constrains or relates them in some way. This construct—declaration list plus optional predicate—has been aptly christened a schema text [Spivey 88]. It can appear in either a horizontal or a vertical form.

h_sch_text	::=	dec_list ⟨ | pred ⟩	13 15
v_sch_text	::=	dec_list ⟨ ST pred_list ⟩	13 14

The vertical form can appear in axiomatic and schema definitions, and the horizontal form can appear in λ, μ and set comprehension terms.
Scope: The scope of the variables introduced in a schema text depends on the type of construct in which the schema text is being used, but it always includes the predicate part of the schema text.

2.3.2 Lists of declarations

Lists of declarations (dec_list) can appear at two levels: local lists are used to introduce variables in schemas, in quantified expressions and in comprehension terms; at a higher level, lists of global declarations introduce constants and operators in definitions.

```
dec_list       ::=  dec ⟨ list_sep dec_list ⟩                13 5 13
                 |  inclusion ⟨ list_sep dec_list ⟩          13 5 13

dec            ::=  decl_id_list :  term                     13 19

decl_id_list   ::=  decl_id ⟨ , decl_id_list ⟩               13 13

decl_id        ::=  id                                       8
                 |  func _ ⟨ rfunc ⟩
                 |  _ pfunc
                 |  ( _ op _ )
                 |  lop _ op _
                 |  _ op _ rop
                 |  _ rel _

inclusion      ::=  schema_ref ⟨ decor ⟩                     9 8
```

Each element of a dec_list is either a global declaration or an inclusion. The name used in the reference of an inclusion must denote a schema. A global identifier is a new constant or a symbol. When a symbol denotes a function, it can be defined to be used in a prefix form (func), a postfix form (pfunc), or an infix form (op). When a symbol denotes a relation, it can be defined to be used in an infix form (rel). Various marker symbols may also be used in combination with func and op symbols: rfunc lop rop. Underscores (_) indicate the positions of the operands. The complete name of the function, operator or relation includes the underscores. However, these are always omitted when operands are supplied. Chapter 4 contains a list of the special symbols which name the prefix and postfix functions (func pfunc), the infix operators (op) and the infix relation symbols (rel) from the Z basic library.
Binding: Operator symbols which are introduced using in a global declaration bind in the

order `pfunc`, `func`, `op`, `rel`. Relational symbols (`rel`) bind more strongly than the logical connectives. Postfix function symbols `pfunc` are left associative i.e. the left-most operator binds most strongly e.g. r^{*-1} is parsed as $(r^*)^{-1}$. Prefix function symbols `func` are right associative i.e. the right-most operator binds most strongly e.g. $\# \bigcup \{S1, S2\}$ is parsed as $\#(\bigcup\{S1, S2\})$. The relative binding powers of infix operator symbols (`op`) are left unspecified.
Scope: If an inclusion is used in a declaration, the local variables of the included schema become known.
Type: A declaration defines a list of variables of the same type. If the term used in a declaration denotes a set, then the value of the variables can only range over that set, and the type of the variables declared is the same as the type of the elements of that set. If the term is a schema, then the type of the variables declared is the schema type associated with that schema.

2.3.3 Lists of predicates

Predicate lists are used as statements about mathematical variables (e.g. in theorems and axiomatic definitions).

```
pred_list    ::=   pred ⟨ list_sep pred_list ⟩                                  15 5 14
```

2.4 Theorems

```
theorem      ::=   ⊢ pred                                                       15
                 | TH ⟨ given_set_def ⟨ list_sep ⟩ ⟩ ⟨ hyps ⟩ ⊢ concl ETH     6 5 15 14
                 | SB ⟨ given_set_def ⟨ list_sep ⟩ ⟩ ⟨ hyps ⟩ ⊢ concl ESB     6 5 15 14

concl        ::=   pred_list                                                    14
```

A theorem may be a turnstile (⊢) followed by a deducible statement (the conclusion) which takes the form of a predicate. If the conclusion is a reference to a schema term, all the variables of the schema term (its signature) must already be declared in the document, and, furthermore, the predicate part (its constraint) must be true. A theorem may also have an hypothesis, in which case the hypothesis and the conclusion are enclosed in theorem brackets (`TH` and `ETH`). In this form, the conclusion may be a list of predicates. New given sets can be introduced in square brackets.

```
hyps          ::=   hyp_list                                   15
                  | schema_term                                24

hyp_list      ::=   h_sch_text ⟨ list_sep hyp_list ⟩           12 5 15
                  | pred ⟨ list_sep hyp_list ⟩                 15 5 15
```

The hypothesis introduces new local variables, which can be introduced either through a hypothesis list or indirectly through a schema term. If the hypothesis is a schema term, then the variables of the schema (its signature) will be introduced and the predicate part (its constraint) will be part of the hypothesis.
Appearance: The TH and ETH tags cause indentation when they are used. For example

> *document...*
>   *dec_list*
> $\vdash$
>   *pred_list*
> *document...*

2.5 Predicates

A predicate is a statement about sets or elements of sets. Predicates are used as statements in axiomatic definitions and global constraints. They are also used in theorems as statements in the hypothesis or conclusion. In schemas, predicates are used to constrain the values taken by local variables. In comprehension terms, they are used in the descriptions of sets, functions and particular elements.

```
pred          ::=   schema_ref                                 9
                  | pred1                                      15

pred1         ::=   ( pred )                                   15
                  | SI pred_list EI                            14
                  | log_exp                                    16
                  | quant_exp                                  17
                  | rel_exp                                    18
```

A predicate may take the following forms:

1. A reference to a schema can be a predicate. If this form is used, all the local variables of the schema must have been declared in the context in which the reference appears. The predicate denoted by the reference is the predicate part of the

referenced schema.

2. Predicates can be grouped by enclosing them in parentheses ( ).

3. Predicates can be grouped by enclosing them in 'indentation brackets' `SI`, `EI` .

4. Predicates can be compound logical expressions (see below).

5. Predicates can be compound quantified expressions (see below).

6. Predicates can be compound relational expressions (see below).

Appearance: Use of `SI`, `EI` (start indentation, end indentation) causes indentation. The predicates of a list which are at the same indentation level are conjoined. For example

document...
$\forall declist\ \cdot$
  $p1$
  $p2$
  $p3$
document...

is parsed as

document...
$\forall declist \cdot (p1) \wedge (p2) \wedge (p3)$
document...

2.5.1 Logical expressions

log_exp	::=	¬ **pred**	15
	|	**pred** ∧ **pred**	15 15
	|	**pred** ∨ **pred**	15 15
	|	**pred** ⇒ **pred**	15 15
	|	**pred** ⇔ **pred**	15 15

A logical expression can be

1. a negation using ¬ (not);

2. a conjunction using the connective ∧ (and);

3. a disjunction using the connective ∨ (or);

4. an implication using the connective ⇒ (implies);

5. an equivalence using the connective $\Leftrightarrow$ (iff—if and only if).

Binding: $\neg$ binds stronger than the binary logical connectives which bind in the order $\wedge, \vee, \Rightarrow, \Leftrightarrow$.
The binary logical connectives are left associative e.g. $(p_1 \Rightarrow p_2 \Rightarrow p_3)$ is the same as $((p_1 \Rightarrow p_2) \Rightarrow p_3)$.

2.5.2 Quantified expressions

```
quant_exp     ::=  ∃ h_sch_text · pred                     12 15
                |  ∃1 h_sch_text · pred                    12 15
                |  pred where h_sch_text                   15 12
                |  pred where SI v_sch_text EI             15 12
                |  ∀ h_sch_text · pred                     12 15
```

A quantified expression can be:

1. an existential quantification using the existential quantifiers ($\exists$, there exists, and $\exists_1$, there exists exactly one), or the **where**keyword; or
2. a universal quantification using the universal quantifier ($\forall$, for all).

The predicate following the vertical bar (|) of the schema text constrains the values of the quantified variables. The predicate following the dot ($\cdot$) is the quantified predicate.
Scope: The schema text in a quantified expression introduces bound variables. These are local to the construct.
Binding: All of the logical connectives $(\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow)$ bind stronger than $\cdot$ (dot) and | (bar), which bind stronger than **where**, which in turn binds stronger than the predicate list separator (;). For example, the predicate list

$$\forall decl \cdot p_1 \wedge p_2 \wedge p_3 ; p_4$$

should be read as

$$(\forall decl \cdot p_1 \wedge p_2 \wedge p_3) ; p_4$$

2.5.3 Relational expressions

```
rel_exp      ::=  term = rel_exp_tail                    19 18
                | term ∈ term                            19 19
                | term term                              19 19
                | term rel rel_exp_tail                  19 18

rel_exp_tail ::=  term ( = rel_exp_tail )                19 18
                | term ( rel rel_exp_tail )              19 18
```

Note: `rel` is not defined in this syntax, but the list of symbols, which is described in chapter 4, indicates which symbols can be used as infix relational symbols.
In a relational expression, a predicate may be constructed using equality (=), membership (∈), or a predefined relational operator. A relational expression may also be constructed from two terms (i.e. juxtaposition). For example, if `S` is a set and `e` is an element, then `S(e)` is equivalent to `e ∈ S`.
Binding: Equality, membership and the relational operators bind stronger than ¬ and the binary logical connectives . For example

$$1 = a \vee \neg\ b \in S$$

should be read as

$$(1 = a) \vee (\neg(b \in S)$$

Type: The components of an equality expression must all be of the same type. If the term on the left of a membership expression is of type `T`, then the term on the right must be of type **P**`T`. The type-correctness of a relational expression is determined by the definition of the the relational operator.

2.6 Terms

Terms are used in syntactic definitions to denote sets or elements of sets. In a declaration, a term is used to denote the set over which the declared variables can range. Terms are also used in relational expressions as statements about elements and sets, and in other terms for constructing new terms.

```
term             ::=  term_reference                        19
                   |  ( term )                              19
                   |  comprehension                         20
                   |  expl_constr                           21
                   |  selection                             22
                   |  construction                          22

term_reference ::=    reference                             8
                   |  schema_ref ⟨ decor ⟩                  9 8
                   |  pre term                              19
                   |  term in term                          19 19
                   |  term post                             19
```

A term can take the following forms:

1. A term can be a reference to a set, an element or a schema. If the term is a `base_name`, possibly with a `decoration`, then it may either be parsed as a `reference` or a `schema_ref`. This ambiguity is resolved by examining the name: if it is a schema name then the latter parse is chosen. The instantiation of generically defined constants takes a special form if the constants have been given symbol names (pre, `in`, `post`). pre binds as `func` and `post` binds as pfunc (see section 2.6.4). in binds less tightly than op and is right associative.

2. Terms can be enclosed in parentheses. Bracketing is used to associate explicitly component terms within constructions. The associativity laws, which allow bracketing to be omitted, are described below.

3. A comprehension term denotes a set or a function (see below).

4. An explicit construction term denotes a tuple, a set or a sequence (see below).

5. A selection term denotes an individual named component of a 'structured' term of schema type (see below).

6. A construction term constructs new sets or elements from existing sets, elements, functions and operators (see below).

Type: If a term is a reference to a schema, the term denotes the schema type associated with the referenced schema (see section 2.7.1. The type of other references is determined by the definition (and the instantiation where relevant) of the referenced object. The type of other terms is described below.

2.6.1 Comprehension terms

```
comprehension ::=  { h_sch_text ( · term ) }      12 19
                 | λ h_sch_text   · term           12 19
                 | μ h_sch_text ( · term )         12 19
```

Scope: The declaration list in the schema text of a comprehension term introduces some bound variables which are local to the construct.
A comprehension can take the following forms:

1. A set comprehension term introduces some bound variables in a declaration list, constrains their values using a predicate, and uses a constructing term to indicate the value of the elements of the set. For example

   $$\{x : \mathbb{N} \mid x \in 1..3 \cdot x + 3\} = \{4, 5, 6\}$$

   If the constructing term is omitted, the elements of the set are tuples, which are constructed from the order of the declarations. For example

   $$\begin{aligned} & \{x : \mathbb{N}; y : \mathbb{N} | x \in 1..2 \land y < x\} \\ = \; & \{x : \mathbb{N}; y : \mathbb{N} | x \in 1..2 \land y < x \cdot (x, y)\} \\ = \; & \{(1,0), (2,1), (2,0)\} \end{aligned}$$

   If there is only one declaration, then the members are single elements. However if the `dec_list` is a single schema inclusion, then the set is composed of the elements of the schema type. For example, if S is a schema

   $$\{S\} = \{S \cdot \theta S\}$$

   Type: The type of the elements of the newly formed set is the type of the constructing term. The type of the new set is the power set of the type of its elements.

2. A lambda abstraction term denotes a function. The bound variables define the parameters of the function. The constraining predicate defines the domain of the function. The constructing term indicates the value of the result. For example

   $$\begin{aligned} \lambda x : \mathbb{N} | x \leq 2 \cdot x + 3 & = \{x : \mathbb{N} | x \leq 2 \cdot x \mapsto (x + 3)\} \\ & = \{(0,3), (1,4), (2,5)\} \\ & = \{0 \mapsto 3, 1 \mapsto 4, 2 \mapsto 5\} \end{aligned}$$

   Type: The type of the function's parameter is the cartesian product of the types of the individual parameters. The type of the result is the type of the constructing term. The function itself is a subset of the cartesian product of its parameters' type and its result's type.

3. A choice term denotes a single element. The value of the element must be uniquely determined by the bound variables, the constraining predicate and the constructing term. For example

$$(\mu\ x, y : \mathbb{N} | y = x + 1 \wedge x - 2 = 5 \cdot (y, x + 4)) = (8, 11)$$

If the term is omitted, the default is the tuple constructed from the declaration, as for set comprehensions above.
Type: The type of a choice term is the type of the constructing term.

2.6.2 Explicit construction terms

```
expl_constr   ::=  tuple                              21
                |  { term_list }                      9
                |  lseq ⟨ term_list ⟩ rseq            21 9 21

tuple         ::=  ( term , term_list )               19 9
                |  θ schema_ref                       9

lseq          ::=  ⟨

rseq          ::=  ⟩
```

An explicit construction may denote:

1. a tuple, which is an explicit construction of a tuple from its components or a θ-construction, in which the reference must denote a schema; the local variables of the schema must be declared in the context where the θ-construction appears, either explicitly or implicitly by schema inclusion;

2. a set extension (enclosed in set braces), which is an explicit construction of a set from its members; the `term_list` cannot be a single schema name as the term { `schema_name` } will be parsed as a comprehension term (see section 2.6.1); or

3. a sequence (enclosed in sequence brackets).

Binding: The list separator (,) binds less strongly than any of the operators in construction terms. Note that parentheses cannot be omitted from a tuple: (a, b, c) and $((a, b), c)$ are different terms.
Type: The type of an explicit tuple construction is the cartesian product of the types of the individual terms. The type of a θ-construction is the schema type associated with the name in the reference. If `S` is a schema, then θ`S` and θ`S'` are of the same type (see

section 2.7.2). The type of a set is the power set of the type of the individual component terms which must all be of the same type. A sequence is a partial function from Natural numbers to the type of the component terms, which must also all be of the same type.

2.6.3 Selection terms

```
selection     ::=   term . id                                              19 8
```

The term in a selection must be of schema type and the identifier must be a variable of the signature of the schema. For example, if

$$S \triangleq [\ a : \mathbb{N}; b : X \mid a > 1\ ] \text{ and } elem : S$$

then

$$elem.b$$

is the b component of *elem*. Note that this is a convenient abbreviation for

$$(\lambda S \cdot b)\ elem$$

Type: The type of a selection term is the type of the component that is being selected. In the above example

elem.a is of type $\mathbb{N}$
elem.b is of type X

2.6.4 Construction terms

```
construction  ::=   product × term                                         23 19
                  | P term                                                 19
                  | func_appl                                              23
                  | term op term ⟨ rop ⟩                                   19 19
                  | lop term op term                                       19 19
                  | func term ⟨ rfunc ⟩                                    19
                  | term pfunc                                             19
```

```
product       ::=    ( product × ) term                                23 19

func_appl     ::=    term term                                         19 19
```

A construction term can take the following forms:

1. A cartesian product. The component terms in a cartesian product must all denote sets.
   Binding: Parentheses cannot be omitted from a cartesian product term.
   Type: The type of a cartesian product is the power set of the cartesian product of the types of the elements of the sets described by the component terms. For example, if we have s : PS and t : PT then s × t is of type P(S × T) since the type of elements of s is S and the type of elements of t is T. The type of the elements of a cartesian product is the cartesian product of the types of the elements of the sets that are described by the component terms. For example, if A and B are given sets, and a: A × B, then a is of type A × B.

2. A power set term. The component term must denote a set.
   Binding: The power set operator ('set of all subsets of') binds stronger than × and all other infix operators.
   Type: The type of a power set term is the power set of the type of the component term. The type of its elements is the same as the type of the component term. For example, if A $\hat{=}$ N \ {0} , and sa: PA, then the type of sa is PN.

3. A function application term. The first component term denotes the function and the second denotes the parameter.
   Binding: The function application separators (space and '(') are left associative i.e. the left-most separator binds most strongly:

   $$f\ a\ b = (f\ a)\ b$$
   $$f\ (a)\ (b) = (f(a))\ (b)$$

   The separator binds stronger than × and the infix operators. For example

   $$f\ a\ +\ b = (f\ a) + b$$

   Prefix and postfix function symbols bind stronger than the function application separators. For example,

   $$f\#S = f\,(\#S)$$

   Type: The type of the parameter supplied must be the same as the type of the elements of the function's domain. The type of the application term is the type of the elements of the function's range.

4. An infix function application term. Infix operators (op) are predefined or defined in definitions. Some infix operators have an end marker, in which case the operator

and the end marker serve as delimiters for the second operand (e.g. R ⦇ S ⦈).
Binding: Infix operator symbols bind stronger than infix relational symbols. The relative binding rules between infix operators are not defined.
Type: The type of infix function application terms is determined as for function application terms. Note that the parameter is the tuple constructed from the two operands.

5. A prefix function application term. Prefix operators (func) are predefined or defined in definitions.
   Binding: Prefix operator symbols bind stronger than infix operator symbols. In sequences of prefix operator symbols, the right-most operator binds most strongly e.g. $\#\bigcup\{S1, S2, S3\}$ is parsed as $\#(\bigcup\{S1, S2, S3\})$.
   Type: As above.

6. A postfix function application term. Postfix operators (pfunc) are predefined or defined in definitions.
   Binding: Postfix operator symbols bind stronger than prefix operator symbols. In sequences of postfix operator symbols, the left-most operator binds most strongly e.g. r^{*-1} is parsed as $(r^{*})^{-1}$.
   Type: As above.

2.7 Schema terms

A schema term is a mathematical structure, similar to a predicate. However a schema term can be given a name and the name can be used in different contexts. The name can be included in a declaration list, it can be used as a predicate, it can be used as a term to denote a type or it can be used as part of a θ-construction.

A schema term is associated with a signature and a constraint. The signature defines the bindable variables and their types, and the constraint limits the values which the bindable variables can take (in any context).

If the name that denotes the schema term is used to denote a predicate or an element of schema type i.e. in a θ-construction, then the bindable variables of the schema term must all be declared in the context.

```
schema_term   ::=   schema_ref                    9
                  | schema                        25
                  | schema_term rename            24 26
                  | ( schema_term )               24
                  | log_sexp                      26
                  | spec_sexp                     27
```

1. a reference to a schema;
2. a schema, which explicitly introduces the signature and constraint of a schema (see below);
3. a renamed schema term, which will rename the bindable variables of the signature of the schema term (see below);
4. a bracketed schema term, which is used to associate schema term components explicitly within schema expressions;
5. a logical schema expression, which is used to construct new schema terms from existing schema terms (see below); or
6. a special purpose schema expression, which is also used to construct new schema terms from existing schema terms (see below).

2.7.1 Schemas

```
schema        ::=   SB v_sch_text ESB                     12
                |   [ h_sch_text ]                        12
```

A schema introduces the signature and constraint of a schema term. It can appear in two forms: a 'box' schema form or a horizontal schema form. The declaration list defines the signature. The variables of the declaration list become bindable variables of the schema, and their type is determined by their declaration. The bindable variables of an included schema (i.e. a schema identifier appearing as an inclusion, in a declaration list, within a schema text) also become bindable variables of the including schema. The constraint of a schema is the conjunction of the predicates in the predicate list, together with the constraint implicitly imposed on each variable by its declaration. If a schema is included, the constraint of that schema is also a constraint of the resulting schema.
Type: A schema can be used to denote a set. This set is the schema type associated with the schema. The type is solely determined by the identifiers used for the bindable variables and their types. The type can be thought of as a set of unordered tuples with named components. The components of such a tuple can only be referred to using the selection construction described in section 2.6.3.

2.7.2 Schema renaming

```
rename        ::=   expl_rename                                    26
                  | decor                                          8

expl_rename   ::=   [ rename_list ]                                26

rename_list   ::=   id / id ( , rename_list )                      8 8 26
```

A schema term can be renamed, the result of which is the renaming of the bindable variables of the schema term. This can be done by renaming, in a list of pairs of identifiers (b / a means b replaces all a's), or by decoration (′ means all variables are decorated with a ′ dash).
Type: If S is a schema, then S and S′ are associated with the same type. If x: S and y: S′ then x and y are of the same type, and θS′ is of the same type as θS. However, if we construct a new schema by the definition T $\widehat{=}$ S′ then the identifiers of the signature of T are different from those in S, so θT is associated with a different type from θS (and hence θS′).

2.7.3 Logical schema expressions

```
log_sexp      ::=   schema_term ∧ schema_term                      24 24
                  | schema_term ∨ schema_term                      24 24
                  | schema_term ⇒ schema_term                      24 24
                  | schema_term ⇔ schema_term                      24 24
                  | ¬ schema_term                                  24
                  | ∃ dec_list · schema_term                       13 24
                  | ∀ dec_list · schema_term                       13 24
```

Logical schema expressions can take the following forms:

1. The logical schema connectives ($\wedge$,$\vee$,$\Rightarrow$,$\Leftrightarrow$) can be used in expressions to construct new schema terms. If a logical schema connective is used, all the bindable variables of the component schema terms become bindable variables of the constructed schema term. The constraint of the constructed schema term consists of the constraints of the component schema terms connected by the appropriate logical connective ($\wedge$ when $\wedge$ is used, $\vee$ when $\vee$ is used, $\Rightarrow$ when $\Rightarrow$ is used, or $\Leftrightarrow$ when $\Leftrightarrow$ is used).

2. If schema negation ($\neg$) is used, the bindable variables stay the same, while the

constraint is negated.

3. Schema quantifiers can also be used to construct new schema terms. The bindable variables of the resulting schema term are those of the quantified schema term which do not appear in the declaration list (i.e. the quantified variables are hidden). The constraint of the resulting schema term is the logical quantification ($\forall$ for $\forall$ and $\exists$ for $\exists$) of the constraint of the quantified schema term, over the variables from the declaration list which is used in the quantification.

Binding: The relative binding power of the logical schema connectives is the same as that of their predicate counterparts.
Type: When the logical schema connectives are used, variables which are common to both schema terms must have the same type. When schema negation is used, care should be taken to normalize the declaration before applying the negation to the predicate part of the schema term. All of the variables in the declaration list of a quantified schema expression must be declared (with the same type) in the quantified schema term.

2.7.4 Special purpose schema expressions

spec_sexp	::=	schema_term $\setminus$ (id_list)	24 8
	|	schema_term ↾ schema_term	24 24
	|	schema_term ⨟ schema_term	24 24
	|	schema_term $\gg$ schema_term	24 24
	|	pre schema_term	24

Special purpose schema expressions may take the following forms:

1. The hiding operator takes a schema term as its first operand and an identifier list as its second operand. The result will be a schema term where the bindable variables are the bindable variables from the first operand excluding the identifiers of the second operand. The constraint of the resulting schema term is the existentially quantified constraint of the first operand, over the variables which are excluded. Since hiding is equivalent to existential schema quantification, there is a constraint on the hidden (ie quantified) variables, as above: the hidden variables must all be contained in the signature of the schema term.

2. The schema projection operator (↾) hides all the components of its first argument except those which are also components of its second argument.

3. The schema composition and schema piping operators both construct a new schema term, and must only be applied to schema terms which describe operations. The semantics of these schema operations are explained in [Woodcock 88].

4. The precondition operator is applied to schema terms which are used in a document to describe operations. The resulting schema is a schema describing the preconditions of the operation. The precise semantics are described in [Woodcock 88].

Binding: Prefix and postfix special purpose schema operators bind stronger than infix special purpose schema operators, which, in turn, bind stronger than logical schema connectives. Individual infix and postfix operators are left associative, while individual prefix operators are right associative.

3 Terminal symbols

In this chapter, we give, for each terminal symbol, the mnemonic, a suggested graphical representation and the name. The following points should be noted:

- Some symbols are tags that are invisible, but are needed in a document for structural information (e.g. `Z`, `EZ`, `NL`).
- Some symbols do not have a mnemonic since they are available on normal ASCII and EBCDIC keyboards (e.g. ; ,).
- In the grammar description of chapter 2, the graphical representations are used, except for symbols which are tags (`Z`, `EZ`, `NL`) and symbols which have a two dimensional graphical representation (`SB`, `ESB` etc).
- The non-terminals `rel`, `func`, `op`, `pfunc` are a collection of terminal symbols and have not been fully defined in the concrete syntax of chapter 2. Some specific terminal symbols have been defined in the basic library for Z, and in chapter 4, we indicate whether a symbol is a member of `rel`, `func`, `op` or `pfunc`.

3.1 Document punctuation

Z		start Z section
EZ		end Z section
NL		new line
	;	semicolon
lsqb	[	left square bracket
rsqb	]	right square bracket

3.2 Identifier lists and identifier symbols

	,	comma
V		version delimiter, start subscript
EV		version delimiter, end subscript
	!	
	?	
	′	
dlr	$	dollar
	=	

The following special symbols are often used in names.

sup		start superscript
esup		end superscript
sub		start subscript
esub		end subscript
Delta	Δ	
Xi	Ξ	
Sigma	Σ	
Pi	Π	

3.3 Definitions and declarations

See also section 3.1

	:	colon
cbar	|	constraint bar
tdef	$\hat{=}$	syntactic equivalence for terms
sdef	$\triangleq$	syntactic equivalence for schema terms
ddef	::=	data type definition
bbar	|	branch separator
lang	$\ll$	left angled bracket for disjoint union
rang	$\gg$	right angled bracket for disjoint union
SR	┌	start vertical rule
ST	├────────	"such that"
ER	└	end vertical rule
SB	see section 3.7	
ESB	see section 3.7	
GE	═══	unique (generic) definition
	_	place holder for generic parameters

3.4 Theorem symbols

See also section 3.7

thrm	$\vdash$	theorem
TH	⌜	start theorem
ETH	⌝	end theorem

3.5 Predicate symbols

	(	left parenthesis
	)	right parenthesis
SI		start indentation
EI		end indentation
not	$\neg$	negation
and	$\wedge$	conjunction
or	$\vee$	disjunction
imp	$\Rightarrow$	implication
iff	$\Leftrightarrow$	equivalence
all	$\forall$	universal quantification
exi	$\exists$	existential quantification
exi1	$\exists_1$	unique existence
where	where	'postfix' existential quantifier
spot	$\cdot$	"such that"
cbar	|	constraint bar
mem	$\in$	"an element of"
	=	equals

3.6 Term symbols

lset	{	left set bracket
rset	}	right set bracket
lambda	λ	lambda abstraction
mu	μ	choice
lseq	$\langle$	start sequence
rseq	$\rangle$	end sequence
	.	selection
prod	$\times$	cartesian product
pset	$\mathbb{P}$	power set
theta	θ	tuple constructor

3.7 Schema notation

lsch	[	left schema bracket
rsch	]	right schema bracket
SB	———	start schema box (after name)

ST	$\vdash$———	
		middle line of schema box
ESB	———————————————	
		end schema box
zand	$\wedge$	schema conjunction
zor	$\vee$	schema disjunction
zimp	$\Rightarrow$	schema implication
zeq	$\Leftrightarrow$	schema equivalence
znot	$\neg$	schema negation
zexi	$\exists$	schema existential quantification
zall	$\forall$	schema universal quantification
zfor	/	renaming
zhide	$\setminus$	schema hiding
zproj	$\upharpoonright$	schema projection
zcmp	$\fatsemi$	schema composition
zpipe	$\gg$	schema piping
pre	**pre**	schema precondition

4 Symbols defined in the Basic Library

This chapter gives a list of symbols and identifiers which are introduced and defined in the Z Basic Library. We shall give a mnemonic for each symbol which is not included in the standard character sets. We shall also indicate, for each symbol, whether it is a relational symbol (rel), an infix operator symbol (op), a prefix function symbol (func) or a postfix function symbol (pfunc). Identifiers denote sets or functions.

4.1 Set notation

	$=$	equals	rel[3]
mem	$\in$	"an element of"	rel[3]
pset	$\mathbb{P}$	power set	func[3]
prod	$\times$	cartesian product	op[3]
neq	$\neq$	not equal to	rel
int	$\cap$	intersection	op
uni	$\cup$	union	op
diff	$\setminus$	set difference	op
subs	$\subseteq$	subset	rel
psubs	$\subset$	proper subset	rel
nem	$\notin$	"not an element of"	rel
dint	$\bigcap$	distributed intersection	func
duni	$\bigcup$	distributed union	func
fset	$\mathbb{F}$	finite subsets	pre
fset1	$\mathbb{F}_1$	non-empty finite subsets	pre
pset1	$\mathbb{P}_1$	non-empty power set	pre
null	$\varnothing$	null set	

4.2 Relation notation

rel	$\leftrightarrow$	relation	op
id		identity relation	
dom		domain of a relation	pre
ran		range of a relation	pre
fcmp	$⨾$	forward relational composition	op
cmp	$\circ$	relational composition	op

[3]These symbols are part of the basic language of Z and are used in the Basic Library to define other symbols.

dres	$\lhd$	domain restriction	op
dsub	⩤	domain subtraction	op
rres	$\rhd$	range restriction	op
rsub	⩥	range subtraction	op
fovr	$\oplus$	functional overriding	op
limg	⦇	left image bracket	op
rimg	⦈	right image bracket	rop
inv	$^{-1}$	inverse	pfunc
iter		iteration	op
riter		end iteration	rop
rtcl	*	reflexive transitive closure	pfunc
tcl	$^{+}$	transitive closure	pfunc
map	$\mapsto$	maps to	op

4.3 Function categories

Some relations have special properties. Sets of relations which show common properties are grouped together and given special names. The names appear as infix operators with sets as operands.

pfun	⇸	partial function	in
tfun	$\rightarrow$	total function	in
ffun	⇻	finite function	in
pinj	⤔	partial injection	in
tinj	$\rightarrowtail$	total injection	in
finj	⤕	finite injection	in
psur	⤀	partial surjection	in
tsur	$\twoheadrightarrow$	total surjection	in
bij	⤖	bijection	in

4.4 Natural numbers

Nat	$\mathbb{N}$	natural numbers	
Int	$\mathbb{Z}$	integers	
	succ	successor	func
	pred	predecessor	func
	min	minimum	
	max	maximum	
	$+$	addition	op

	*	multiplication	op
	−	subtraction	op
	div	division	op
	mod	modulus	op
geq	⩾	greater than or equal to	rel
leq	⩽	less than or equal to	rel
	>	greater than	rel
	<	less than	rel
	..	number interval	op
	#	cardinality	func

4.5 Sequence and bag notation

seq		set of sequences	pre
seq_1		set of non-empty sequences	pre
lseq	⟨	left sequence bracket	
rseq	⟩	right sequence bracket	
front			
last			
head			
tail			
next			
rev			
prefix			
suffix			
squash			
cat	⁀	sequence concatenation	op
dovr	⊕/	distributed overriding	func
dcmp	⨾/	distributed composition	func
dcat	⁀/	distributed concatenation	func
ires	↿	index restriction	op
sres	↾	sequence restriction	op
bag		set of bags	pre
count			
in			
buni	⊎	bag union	op
items			

5 An Abstract Syntax for the Z Notation

An abstract syntax for the Z notation is presented. It is intended to avoid details of concrete syntax: rather than describing a language by defining which strings of characters are permissible, an abstract syntax defines instead objects whose structures are permissible. These objects are usually viewed as trees, and each may abstract many different strings in the concrete syntax. An abstract syntax is therefore more concise and so more readable.

The abstract syntax is described using the Z notation itself. Readers should be warned that this description is not intended for newcomers to the notation. It will be found useful by those who already have some familiarity with Z, and now feel that an abstract view of the notation's syntax would be worthwhile. It is expected that this syntax will prove most useful to people who might be constructing tools for manipulating the language. As has been noted elsewhere [Spivey 85], if we are to use a description of the Z notation—the abstract syntax in this case—as part of the specification of a software tool to manipulate and reason about Z, then we ought to use a notation for that description which is intended for expressing software specifications. The exercise is also a good demonstration of the applicability of Z to a problem such as that of describing syntax abstractly.

In what follows, definitions of syntactic categories are given using data type definitions to express alternatives. For example,

```
BRANCH ::=
     constant ≪ ID ≫
   | constructor ≪ FUNCTION_IMAGE ≫
```

describes a syntactic category BRANCH, a representation of which can be either a constant drawn from the syntactic category ID, or a constructor drawn from the syntactic category FUNCTION_IMAGE, which could be defined using the schema definition

```
FUNCTION_IMAGE ≙ [ function_name: ID; domain: TERM ]
```

This says that a constructor has two components: function_name, drawn from the syntactic category ID, and domain, drawn from the syntactic category TERM. A schema is used to emphasise that in the abstract syntax we really don't care about the order of these components. This allows greater freedom in the design of concrete syntaxes.

The description of the syntax in the Z notation has something of the flavour of a description in Backus-Naur Form, both being tree-like descriptions. However, we can be more

abstract—and more rigorous—in Z; we can avoid unnecessary detail such as ordering; and we can give components convenient names. A similar use of an abstract syntax notation may be found in [Jones 80].

5.1 A document

We first define a construct that will be useful in the following specification: we model an optional construct by a set which has either 0 or 1 elements.

$$\texttt{optional [X]} \triangleq \{\texttt{s: } \mathbb{P}\texttt{X} \mid \texttt{\#s} \leq \texttt{1}\}$$

The specification uses definitions taken from the basic library:

```
BASIC_LIB
```

It is important to understand that a document exists in an environment of named documents.

$$\texttt{Library} \triangleq \texttt{NAME} \nrightarrow \texttt{DOCUMENT}$$

where a `DOCUMENT` is defined as follows:

$$\texttt{DOCUMENT} \triangleq \texttt{[givensets: seq NAME; contents: seq SECTION]}$$

and

```
SECTION ::=
      definition ≪DEF ≫
    | consequence ≪THEOREM ≫
    | import ≪IMPORT ≫
```

A document is generic with respect to some given named sets; a document defines some global variables—these may be either explicit definitions of new variables whose scope is then the entire document, or generic extensions to the language, e.g. definitions of new operators; a document contains some consequences, which are theorems about the definitions given in the document; documents from the library may be imported.

When a document is imported from the library, its variables may be renamed and its generic parameters instantiated:

```
IMPORT ≙ [ doc: NAME; version: optional[NAME]; inst: INST ]

INST ::=
     key ≪ID ⇸ TERM≫
   | position ≪seq TERM≫
```

A document construction contains either one or two NAMEs. The first of these is the name given to the document in the library. The second, if present, is the version decoration. Instantiation may be either by keyword or by position (but not a mixture of the two).

5.2 Identifiers, names and references

In this specification, we assume the existence of a set of names:

```
[NAME]
```

Mathematical variables are named using identifiers, which consist of a "base" name, together with a sequence[3] of decorations:

```
ID ≙ [ basename: NAME; decor: seq DECOR ]

DECOR ::= exclam | query | dash | version ≪NAME≫
```

Mathematical variables are referred to using an identifier, together with an optional document qualifier and an optional instantiation of the identifier's generic parameters. Schema references are very similar, but they cannot contain any decorations.

```
REF ≙ [ id: ID; qual: optional[NAME]; inst: INST ]

SCH_REF ≙ [ name: NAME; qual: optional[NAME]; inst: INST ]
```

[3]It should be noted that seq DECOR includes the *empty* sequence of decorations—we model an undecorated identifier as one with an empty sequence of decorations. Similar use is made of seq, $\mathbb{F}$ and ⤔ below.

5.3 Definitions and declarations

A definition consists of: some declarations together with a predicate which the newly introduced variables must satisfy, or a definition of a new variable by syntactic equivalence, or a definition of a new data type, or a definition of a new schema

```
DEF ::=
      axiomatic ≪AX_DEF≫
    | syntactic ≪SYN_DEF≫
    | datatype ≪DT_DEF≫
    | schemadef ≪SCH_DEF≫
```

An axiomatic definition introduces some constants or operators, together with their types, in the declaration part of a schema text. A predicate may also be used to constrain the newly-introduced variables. An axiomatic definition may be generic.

```
AX_DEF ≙ [ SCHEMA_TEXT ; genparams: seq NAME ]

SCHEMA_TEXT ≙ [ decls: F DEC; property: PRED ]
```

A syntactic definition introduces a new variable and specifies the term to which it is to be equivalent. Syntactic definitions may also be generic.

```
SYN_DEF ≙
      [ var: ID; spec: TERM; genparams: seq NAME ]
```

A data type definition consists of the name of the new data type, together with at least one 'branch' of the tree to be defined. Each branch is either a simple identifier or a constructor function.

```
DT_DEF ≙ [ id: ID; branches: F₁ BRANCH ]

BRANCH ::=
      constant ≪ID≫
    | constructor ≪FUNCTION_IMAGE≫

FUNCTION_IMAGE ≙ [ function_name: ID; domain: TERM ]
```

A schema definition introduces either a single new schema, or a generic family of schemas,

and names them.

```
SCH_DEF ≙
    [ sch_name: NAME; schema: SCHEMATERM; genparams: seq NAME]
```

Within a schema text, variables can be introduced in two ways: either by naming them explicitly (and giving the sets from which they must take their values), or by giving the name of a schema which includes their definition.

```
DEC ::=
    decl ≪VAR_INTRO ≫
  | inclusion ≪SCH_INCL ≫

VAR_INTRO ≙ [ var: ID; range: TERM ]

SCH_INCL ≙ [ ref: SCH_REF; decor: seq DECOR ]
```

5.4 Theorems

A theorem can be generic; it introduces a new context in which the conclusion is to be proved. Furthermore, schemas may be interpreted as theorems.

```
THEOREM ≙ [ givensets: seq NAME; hyp: HYP; conc: PRED ]

HYP ::=
    varhyp ≪SCHEMA_TEXT ≫
  | schhyp ≪SCHEMATERM ≫
```

5.5 Predicates

A predicate can be
1) a schema can be interpreted as a predicate
2) a negated predicate
3) a binary operator from the predicate calculus
4) a quantified predicate, which consists of the quantified variables, a predicate which constrains the values of these variables, and the quantified predicate itself
5) a statement relating two terms
6) 'true'

```
PRED ::=
      predinterp ≪ REF ≫
    | negation ≪ PRED ≫
    | conjunction ≪PARAMS ≫
    | disjunction ≪PARAMS ≫
    | implication ≪PARAMS ≫
    | equivalence ≪PARAMS ≫
    | universalquant ≪QUANT_EXP ≫
    | existquant ≪QUANT_EXP ≫
    | uniqueexist ≪QUANT_EXP ≫
    | binaryrel ≪REL_EXP ≫
    | trueval

PARAMS ≙  [ op1,op2:  PRED ]

REL_EXP ≙  [ op1,op2:  TERM; operator:  RELOP ]

RELOP ::= member | equal | defrelop ≪NAME ≫

QUANT_EXP ≙  [ vars:  SCHEMA_TEXT; pred:  PRED ]
```

5.6 Terms

Terms may take the following forms:
1) term identifiers denote sets or elements. If these sets or elements are generically defined, actual values for the generic parameters can be given.
2) a schema name can be used to denote a set (the schema type) or a tuple.
3) a comprehension form, which introduces some local variables, constrains their values and constructs new terms, can be used in the definition of a set, an element (choice) or a function (lambda).
4) sets, sequences and tuples can be constructed by explicitly naming their elements.
5) a selection term is used to identify components of a schema type.
6) the cartesian product of two or more sets.
7) the set of all subsets of a set.
8) function application.

```
TERM ::=
     termidentifier ≪REF ≫
   | setinterp ≪ SCH_INCL ≫  |  tupleinterp ≪ REF ≫
   | setcomp ≪ COMP ≫  |  lambda ≪ COMP ≫  |  choice ≪ COMP ≫
   | setconstruction ≪F TERM ≫
   | seqconstruction ≪seq TERM ≫
   | tupleconstruction ≪seq TERM ≫
   | selection ≪ SELECT ≫
   | cartprodnct ≪seq TERM ≫
   | powerset ≪ TERM ≫
   | functionappln ≪APPLY ≫
```

where

```
COMP ≙ [ decl: SCHEMA_TEXT; construct: TERM ]

SELECT ≙ [ argument: TERM; selector: ID ]

APPLY ≙ [ func, arg: TERM ]
```

5.7 Schema terms

A schema term can be
1) a reference to a schema
2) a schema which introduces some new variables and constrains their values
3) a schema with its variables renamed
4) a decorated schema
5) the application of a schema function
6) the application of a schema quantifier
7) the hiding of some of the variables of a schema term

```
SCHEMATERM ::=
     schemaname ≪ SCH_REF ≫
   | schema ≪ SCHEMA_TEXT ≫
   | schema_rename ≪ RENAME_SCHEMA ≫
   | schema_decor ≪ DECOR_SCHEMA ≫
   | schemaexp ≪ SCH_EXP ≫
   | schemaquant ≪ QUANT_SCHEMA ≫
   | schemavarhide ≪ HIDE_SCHEMA ≫

RENAME_SCHEMA ≙ [ schema: SCHEMATERM; rename: ID ⤔ ID ]

DECOR_SCHEMA ≙ [ schema: SCHEMATERM; decor: seq DECOR ]

SCH_EXP ≙ [ function: SCHEMA_FUNC; operands: seq SCHEMATERM ]

SCHEMA_FUNC ::=
     not | and | or | imply | equiv | comp | pipe | proj

QUANT_SCHEMA ≙
      [ decl: 𝔽₁ DEC; schema: SCHEMATERM; quantifier: QUANT ]

QUANT ::= universal | existential

HIDE_SCHEMA ≙ [ schema: SCHEMATERM; vars: 𝔽 ID ]
```

Acknowledgments

This document has benefited from contributions from John Nicholls, Steve Powell, and John Wordsworth (IBM), Brian Monahan (IST plc), Trevor King, Colin O'Halloran and Chris Sennett (RSRE), Rod Bark and Sam Valentine (Systems Designers), and Jonathan Bowen, Ian Hayes, Joy Reed, Jane Sinclair, Mike Spivey and Bernard Sufrin of the PRG.

References

[Abrial 81] J-R. Abrial, "A Course on System Specification", Lecture Notes, Programming Research Group, University of Oxford, 1981. (Out of print)

[Hayes 87] I. J. Hayes, ed., *Specification Case Studies*, Prentice-Hall International, 1987.

[Jones 80] C. B. Jones, *Software Development—A Rigorous Approach*, Prentice-Hall International, 1980.

[Jones 86] C. B. Jones, *Systematic Software Development using VDM*, Prentice-Hall International, 1986.

[Morgan 84] C. C. Morgan, "Schemas in Z: a Preliminary Reference Manual", Programming Research Group, University of Oxford, 1984.

[Sørensen 82] I. H. Sørensen, "A Specification Language", in *Program Specification* (J. Staunrup, ed.), Lecture Notes in Computer Science, vol. 134, Springer-Verlag, 1982.

[Spivey 85] J. M. Spivey, "Understanding Z: A Specification Language and its Formal Semantics", DPhil Thesis, Programming Research Group, University of Oxford, 1985.

[Spivey 88] J. M. Spivey, "The Z Notation—A Reference Manual", JMS-87-12d, Programming Research Group, University of Oxford, 1988.

[Sufrin 81] B. A. Sufrin, "Formal Specification: Notation and Examples", in *Tools and Notations for Program Construction* (D. Néel, ed.). Cambridge University Press, 1981.

[Sufrin 86] B. A. Sufrin, ed., "Z Handbook, Draft 1.1", Programmiug Research Group, University of Oxford, 1986.

[Woodcock 88] J. C. P. Woodcock, "Structuring Specifications", Programming Research Group, University of Oxford, 1988.

Index of definitions

axiomatic definition 9
base name 8
bindable variable 24
bound variable 17
cartesian product 23
choice 21
comprehension 19
conjunction 16
constant 6
constraint 24
constructing term 20
construction 19
data type definition 11
data type 6
decoration 8
definition 6
disjunction 16
document qualifier 9
document 5
equality 18
equivalence 17
existential quantifiers 17
explanatory text 5
explicit construction 19
function application 23
given set definition 6
given set 6
global constraint 7
global declaration 13
global variable 7
hiding operator 27
identifier 8
if and only if 17
implication 16
import 7
inclusion 13
infix function application 23
lambda abstraction 20
logical connective 17
logical schema connective 26
logical schema expression 25

membership 18
negation 16
number 8
postfix function application 24
power set 23
precondition operator 28
predicate list 14
predicate 15
prefix function application 24
quantified expression 17
quantified predicate 17
quantified variable 17
relational expression 18
relational operator 18
schema composition 27
schema definition 12
schema negation 26
schema piping 27
schema projection 27
schema quantifier 27
schema term 24
schema text 12
schema type 19
schema 6
selection 19
sequence 21
set comprehension 20
set extension 21
signature 24
special purpose schema expression 25
syntactic definition 10
term 18
theorem 7
θ-construction 21
tuple 21
universal quantifier 17
where 17
Z phrase 5
Z text 5

Summary of Syntax

```
document        ::= explan_text ( document )
                  | Z z_text EZ ( document )

explan_text     ::= "characters, excluding Z and EZ"

z_text          ::= ( z_phrase ) ( list_sep z_text )

list_sep        ::= ;
                  | NL

z_phrase        ::= given_set_def
                  | definition
                  | constraint
                  | theorem
                  | import

given_set_def   ::= [ name_list ]

definition      ::= axiomatic_def
                  | syntactic_def
                  | datatype_def
                  | schema_def

constraint      ::= pred1

import          ::= base_name ( version )
                              ( instantiation )
```

```
id_list         ::= id ( , id_list )

id              ::= base_name ( decor )

base_name       ::= "alphanumerics, underscores and
                    symbols, excluding terminal symbols
                    and operator symbols"

name_list       ::= base_name ( , name_list )

decor           ::= version ( decor )
                  | attribute ( decor )

version         ::= V base_name EV

attribute       ::= !
                  | ?
                  | '

reference       ::= ( doc_qual $ ) id ( instantiation )

doc_qual        ::= base_name ( version )

instantiation   ::= [ inst_list ]

inst_list       ::= term_list
                  | binding_list

term_list       ::= term ( , term_list )

binding_list    ::= id = term ( , binding_list )

schema_ref      ::= ( doc_qual $ ) base_name
                              ( instantiation )
```

```
axiomatic_def   ::= liberal_def
                  | generic_def

liberal_def     ::= dec ( | pred )
                  | SB v_sch_text ER

generic_def     ::= decl_id params : term ( | pred )
                  | BR ( params ) GE v_sch_text ER

params          ::= [ name_list ]

syntactic_def   ::= decl_id ( params ) ≙ term
                  | const_sym ≙ term

const_sym       ::= pre id
                  | id post
                  | id in id

datatype_def    ::= id ::= branches

branches        ::= id ( ⟪ term ⟫ ) ( | branches )

schema_def      ::= base_name ( params ) ≙ schema_term
                  | base_name ( params ) SB v_sch_text ESB

h_sch_text      ::= dec_list ( | pred )

v_sch_text      ::= dec_list ( ST pred_list )

dec_list        ::= dec ( list_sep dec_list )
                  | inclusion ( list_sep dec_list )

dec             ::= decl_id_list : term

decl_id_list    ::= decl_id ( , decl_id_list )

decl_id         ::= id
                  | func _ ( rfunc )
                  | _ pfunc
                  | ( _ op _ )
                  | lop _ op _
                  | _ op _ rop
                  | _ rel _

inclusion       ::= schema_ref ( decor )

pred_list       ::= pred ( list_sep pred_list )
```

```
theorem         ::= ⊢ pred
                  | TH ( given_set_def ( list_sep ) )
                              ( hyps ) ⊢ concl ETH
                  | SB ( given_set_def ( list_sep ) )
                              ( hyps ) ⊢ concl ESB

concl           ::= pred_list

hyps            ::= hyp_list
                  | schema_term

hyp_list        ::= h_sch_text ( list_sep hyp_list )
                  | pred ( list_sep hyp_list )
```

```
pred            ::=  schema_ref
                  |  pred1

pred1           ::=  ( pred )
                  |  SI pred_list EI
                  |  log_exp
                  |  quant_exp
                  |  rel_exp

log_exp         ::=  ¬ pred
                  |  pred ∧ pred
                  |  pred ∨ pred
                  |  pred ⇒ pred
                  |  pred ⇔ pred

quant_exp       ::=  ∃ h_sch_text · pred
                  |  ∃1 h_sch_text · pred
                  |  pred where h_sch_text
                  |  pred where SI v_sch_text EI
                  |  ∀ h_sch_text · pred

rel_exp         ::=  term = rel_exp_tail
                  |  term ∈ term
                  |  term term
                  |  term rel rel_exp_tail

rel_exp_tail    ::=  term ( = rel_exp_tail )
                  |  term ( rel rel_exp_tail )
```

```
term            ::=  term_reference
                  |  ( term )
                  |  comprehension
                  |  expl_constr
                  |  selection
                  |  construction

term_reference  ::=  reference
                  |  schema_ref ( decor )
                  |  pre term
                  |  term in term
                  |  term post

comprehension   ::=  { h_sch_text ( · term ) }
                  |  λ h_sch_text · term
                  |  μ h_sch_text ( · term )

expl_constr     ::=  tuple
                  |  { term_list }
                  |  lseq ( term_list ) rseq

tuple           ::=  ( term , term_list )
                  |  θ schema_ref

lseq            ::=  ⟨

rseq            ::=  ⟩

selection       ::=  term . id

construction    ::=  product × term
                  |  P term
                  |  func_appl
                  |  term op term ( rop )
                  |  lop term op term
                  |  func term ( rfunc )
                  |  term pfunc

product         ::=  ( product × ) term

func_appl       ::=  term term
```

```
schema_term     ::=  schema_ref
                  |  schema
                  |  schema_term rename
                  |  ( schema_term )
                  |  log_sexp
                  |  spec_sexp

schema          ::=  SB v_sch_text ESB
                  |  [ h_sch_text ]

rename          ::=  expl_rename
                  |  decor

expl_rename     ::=  [ rename_list ]

rename_list     ::=  id / id ( , rename_list )

log_sexp        ::=  schema_term ∧ schema_term
                  |  schema_term ∨ schema_term
                  |  schema_term ⇒ schema_term
                  |  schema_term ⇔ schema_term
                  |  ¬ schema_term
                  |  ∃ dec_list · schema_term
                  |  ∀ dec_list · schema_term

spec_sexp       ::=  schema_term \ ( id_list )
                  |  schema_term ↾ schema_term
                  |  schema_term ⨾ schema_term
                  |  schema_term ≫ schema_term
                  |  pre schema_term
```