BRITISH STANDARD

BS ISO/IEC 13568:2002

Information technology — Z formal specification notation — Syntax, type system and semantics

ICS 35.060

NO COPYING WITHOUT BSI PERMISSION EXCEPT AS PERMITTED BY COPYRIGHT LAW

Document Sharing Hub
British Standards

Full document details: BS ISO/IEC 13568:2002

National foreword

This British Standard reproduces verbatim ISO/IEC 13568:2002 and implements it as the UK national standard.

The UK participation in its preparation was entrusted to Technical Committee IST/5, Programming languages, which has the responsibility to:

— aid enquirers to understand the text;

— present to the responsible international/European committee any enquiries on the interpretation, or proposals for change, and keep the UK interests informed;

— monitor related international and European developments and promulgate them in the UK.

A list of organizations represented on this committee can be obtained on request to its secretary.

Cross-references

The British Standards which implement international publications referred to in this document may be found in the *BSI Catalogue* under the section entitled "International Standards Correspondence Index", or by using the "Search" facility of the *BSI Electronic Catalogue* or of British Standards Online.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for their correct application.

Compliance with a British Standard does not of itself confer immunity from legal obligations.

This British Standard, having been prepared under the direction of the DISC Board, was published under the authority of the Standards Policy and Strategy Committee on 8 August 2002

Summary of pages

This document comprises a front cover, an inside front cover, the ISO/IEC title page, pages ii to vi, pages 1 to 189, and a back cover.

The BSI copyright date displayed in this document indicates when the document was last issued.

Amendments issued since publication

Amd. No.	Date	Comments

© BSI 8 August 2002

ISBN 0 580 40198 7

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

INTERNATIONAL STANDARD

ISO/IEC 13568

First edition
2002-07-01

Information technology — Z formal specification notation — Syntax, type system and semantics

Technologies de l'information — Notation Z pour la spécification formelle — Syntaxe, système de caractères et sémantique

Reference number
ISO/IEC 13568:2002(E)

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

Contents

Page

Foreword v

Introduction vi

1 Scope 1

2 Normative references 1

3 Terms and definitions 1

4 Metalanguages 3

5 Conformance 15

6 Z characters 18

7 Lexis 24

8 Concrete syntax 30

9 Characterisation rules 39

10 Annotated syntax 40

11 Prelude 43

12 Syntactic transformation rules 44

13 Type inference rules 55

14 Semantic transformation rules 66

15 Semantic relations 71

Annex A (normative) Mark-ups 79

Annex B (normative) Mathematical toolkit 94

Annex C (normative) Organisation by concrete syntax production 111

Annex D (informative) Tutorial 158

Annex E (informative) Conventions for state-based descriptions 173

Bibliography 175

Index 176

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

Figures

1 Phases of the definition . . . 16
B.1 Parent relation between sections of the mathematical toolkit . . . 94
D.1 Concrete parse tree of birthday book example . . . 161
D.2 Tree of birthday book example after syntactic transformation . . . 164
D.3 Annotated tree of axiomatic example . . . 165
D.4 Annotated tree of generic example . . . 168
D.5 Annotated tree of chained relation example . . . 172

Tables

1 Syntactic metalanguage . . . 3
2 Parentheses in metalanguage . . . 4
3 Propositional connectives in metalanguage . . . 5
4 Quantifiers in metalanguage . . . 5
5 Abbreviations in quantifications in metalanguage . . . 5
6 Conditional expression in metalanguage . . . 5
7 Propositions about sets in metalanguage . . . 6
8 Basic set operations in metalanguage . . . 6
9 Powerset in metalanguage . . . 7
10 Operations on natural numbers in metalanguage . . . 7
11 Decorations of names in metalanguage . . . 7
12 Tuples and Cartesian products in metalanguage . . . 8
13 Function comprehensions in metalanguage . . . 8
14 Relations in metalanguage . . . 8
15 Proposition about relations in metalanguage . . . 8
16 Functions in metalanguage . . . 9
17 Application in metalanguage . . . 9
18 Sequences in metalanguage . . . 9
19 Disjointness in metalanguage . . . 9
20 Metavariables for phrases . . . 10
21 Metavariables for operator words . . . 11
22 Environments . . . 12
23 Metavariables for environments . . . 12
24 Variables over type universe . . . 12
25 Type relations . . . 12
26 Type sequents . . . 13
27 Semantic universe . . . 14
28 Variables over semantic universe . . . 14
29 Semantic relations . . . 14
30 Semantic idioms . . . 15
31 Operator precedences and associativities . . . 37

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this International Standard may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 13568 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Annexes A to C form a normative part of this International Standard. Annexes D and E are for information only.

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

Introduction

This International Standard specifies the syntax, type system and semantics of the Z notation, as used in formal specification.

A specification of a system should aid understanding of that system, assisting development and maintenance of the system. Specifications need express only abstract properties, unlike implementations such as detailed algorithms, physical circuits, etc. Specifications may be loose, allowing refinement to many different implementations. Such abstract and loose specifications can be written in Z notation.

A specification written in Z notation models the specified system: it names the components of the system and expresses the constraints between those components. The meaning of a Z specification—its semantics—is defined as the set of interpretations (values for the named components) that are consistent with the constraints.

Z uses mathematical notation, hence specifications written in Z are said to be formal: the meaning is captured by the form of the mathematics used, independent of the names chosen. This formal basis enables mathematical reasoning, and hence proofs that desired properties are consequences of the specification. The soundness of inference rules used in such reasoning should be proven relative to the semantics of the Z notation.

This International Standard establishes precise syntax and semantics for a system of notation for mathematics, providing a basis on which further mathematics can be formalized.

Particular characteristics of Z include:

- its extensible toolkit of mathematical notation;
- its schema notation for specifying structures in the system and for structuring the specification itself; and
- its decidable type system, which allows some well-formedness checks on a specification to be performed automatically.

Examples of the kinds of systems that have been specified in Z include:

- safety critical systems, such as railway signalling, medical devices, and nuclear power systems;
- security systems, such as transaction processing systems, and communications; and
- general systems, such as programming languages and floating point processors.

Standard Z will also be appropriate for use in:

- formalizing the semantics of other notations, especially in standards documents.

This is the first ISO standard for the Z notation. Much has already been published about Z. Most uses of the Z notation have been based on the examples in the book "Specification Case Studies" edited by Hayes [2][3]. Early definitions of the notation were made by Sufrin [14] and by King *et al* [8]. Spivey's doctoral thesis showed that the semantics of the notation could be defined in terms of sets of models in ZF set theory [11]. His book "The Z Notation—A Reference Manual" [12][13] is the most complete definition of the notation, prior to this International Standard. Differences between Z as defined here and as defined in [13] are discussed in [15]. This International Standard addresses issues that have been resolved in different ways by different users, and hence encourages interchange of specifications between diverse tools. It also aims to be a complete formal definition of Z.

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

Information technology— Z formal specification notation— Syntax, type system and semantics

1 Scope

The following are within the scope of this International Standard:

- the syntax of the Z notation;
- the type system of the Z notation;
- the semantics of the Z notation;
- a toolkit of widely used mathematical operators;
- LaTeX [10] and e-mail mark-ups of the Z notation.

The following are outside the scope of this International Standard:

- any method of using Z, though an informative annex (E) describes one widely-used convention.

2 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this International Standard. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

ISO/IEC 10646-1:2000, *Information technology—Universal Multiple-Octet Coded Character Set (UCS)—Part 1: Architecture and Basic Multilingual Plane, with amendment 1, with its amendments and corrigenda*

ISO/IEC 10646-2:2001, *Information technology—Universal Multiple-Octet Coded Character Set (UCS)—Part 2: Supplementary Planes*

ISO/IEC 14977:1996, *Information technology—Syntactic metalanguage—Extended BNF*

3 Terms and definitions

For the purposes of this International Standard, the following terms and definitions apply. *Italicized* terms in definitions are themselves defined in this list.

3.1
binding
finite function from names to values

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

3.2
capture
cause a reference expression to refer to a different declaration from that intended

3.3
carrier set
set of all values in a type

3.4
constraint
property that is either true or false

3.5
environment
function from names to information used in type inference

3.6
interpretation
function from global names of a section to values in the *semantic universe*

3.7
metalanguage
language used for defining another language

3.8
metavariable
name denoting an arbitrary phrase of a particular syntactic class

3.9
model
interpretation that makes the defining *constraints* of the corresponding section be true

3.10
schema
set of *bindings*

3.11
scope of a declaration
part of a specification in which a reference expression whose name is the same as a particular declaration refers to that declaration

3.12
scope rules
rules determining the *scope of a declaration*

3.13
semantic universe
set of all semantic values, providing representations for both non-generic and generic Z values

3.14
signature
function from names to types

3.15
type universe
set of all type values, providing representations for all Z types

3.16
Z core language
Z notation defined by this International Standard excepting the notation of the mathematical toolkit

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

3.17
ZF set theory
Zermelo-Fraenkel set theory

4 Metalanguages

The metalanguages are the notations used to define the syntax, type system and semantics of Z.

4.1 Syntactic metalanguage

The syntactic metalanguage used is the subset of the standard ISO/IEC 14977:1996 [7] summarised in Table 1, with modifications so that the mathematical symbols of Z can be presented in a more comprehensible way.

Table 1 — Syntactic metalanguage

Symbol	Definition
`=`	defines a non-terminal on the left in terms of the syntax on the right.
`|`	separates alternatives.
`,`	separates notations to be concatenated.
—	separates notation on the left from notation to be excepted on the right.
`{ }`	delimit notation to be repeated zero or more times.
`[ ]`	delimit optional notation.
`( )`	are grouping brackets (parentheses).
`' '`	delimit a terminal symbol.
`;`	terminates a definition.
`? ?`	delimit informal definition of notation.
(*)*	delimit commentary.

The infix operators `|` and `,` have precedence such that parentheses are needed when concatenating alternations, but not when alternating between concatenations. The exception notation is always used with parentheses, making its precedence irrelevant. Whitespace separates tokens of the syntactic metalanguage; it is otherwise ignored.

EXAMPLE The lexis of a `NUMERAL` token, and its informal reading, are as follows.

```
NUMERAL          =  NUMERAL , DECIMAL
                 |  DECIMAL
                 ;
```

The non-terminal symbol `NUMERAL` stands for a maximal sequence of one or more decimal digit characters (without intervening white space).

The changes to ISO/IEC 14977:1996 allow use of mathematical symbols in the names of non-terminals, and are formally defined as follows.

```
Meta identifier character  =  ? all cases from ISO/IEC 14977:1996 ?
                           |  '(' | ')' | '[' | ']' | '{' | '}' | '⦇' | '⦈' | '⟪' | '⟫'
                           |  'ℙ' | ':' | ',' | '|' | '&' | '\' | '/' | '.' | ';' | '_' | '='
                           |  '?' | '⊢' | '∀' | '•' | '∃' | '⇔' | '⇒' | '∨' | '∧' | '¬'
                           |  '∈' | '↾' | '×' | 'λ' | 'μ' | 'θ' | '⨾' | '≫' | '⨟'
                           ;
```

NOTE This anticipates a future version of ISO/IEC 14977:1996 permitting use of the characters of ISO/IEC 10646 [5][6].

The following standard metalanguage characters have been overloaded as `Meta identifier characters`: `'='`, `'|'`, `','`, `'{'`, `'}'`, `'['`, `']'`, `'('`, `')'` and `';'`. Uses of them as `Meta identifiers` are with the common

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

suffix -tok, e.g. (-tok, which may be viewed as a postfix metalanguage operator. Uses of them as meta-operators are further distinguished by preceding and following these uses with space.

A further change to ISO/IEC 14977:1996 is the use of multiple fonts: metalanguage characters and non-terminals are in `typewriter`, those non-terminals that correspond to Z tokens appear as those Z tokens normally appear, typically in Roman, and comments are in *italic*.

The syntactic metalanguage is used in defining Z characters, lexis, concrete syntax and annotated syntax (clauses 6, 7, 8 and 10).

4.2 Mathematical metalanguage

4.2.1 Introduction

Logic and ZF set theory are the basis for the semantics of Z. In this section the specific notations used are described. The notations used here are deliberately similar in appearance to those of Z itself, but are grounded only on the logic and set theory developed by the wider mathematical community.

The mathematical metalanguage is used in type inference rules and in semantic relations (clauses 13 and 15).

4.2.2 Parentheses

The forms of proposition and expression are given below. Where there could be any ambiguity in the parsing, usually parentheses have been used to clarify, but in any other case the precedence conventions of Z itself are intended to be used.

The use of parentheses is given in tabular form in Table 2, where p stands for any proposition and e stands for any expression.

Table 2 — Parentheses in metalanguage

Notation	Definition
(p)	p
(e)	e

The same brackets symbols are used around pairs and tuple extensions (Table 12), but those cannot be omitted in this way.

4.2.3 Propositions

4.2.3.1 Introduction

The value of a metalanguage proposition is either *true* or *false*. The values *true* and *false* are distinct. In this International Standard, no proposition of the metalanguage is both *true* and *false*; that is, this metatheory is consistent. Furthermore, every proposition is either *true* or *false*, even where it is not possible to say which; that is, the logic is two-valued.

4.2.3.2 Propositional connectives

The propositional connectives of negation, conjunction and disjunction are used. In Table 3 and later, p, p_2, etc, represent arbitrary propositions.

Conjunction is also sometimes indicated by writing propositions on successive lines, as a vertical list.

4.2.3.3 Quantifiers

Existential, universal and unique-existential quantifiers are used. In Tables 4 and 5 and later, i, i_2, etc, represent arbitrary names, e, e_2, etc, represent arbitrary expressions, and ... represents zero or more repetitions of the surrounding formulae; in these tables, the propositions can contain references to the names, but the expressions cannot.

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

4 Metalanguages

Table 3 — Propositional connectives in metalanguage

Notation	Name	Definition
$\neg\ p$	negation	*true* iff p is *false*
$p_1 \wedge p_2$	conjunction	*true* iff p_1 and p_2 are both *true*
$p_1 \vee p_2$	disjunction	*false* iff p_1 and p_2 are both *false*

Table 4 — Quantifiers in metalanguage

Notation	Name	Definition
$\exists\ i_1 : e_1;\ ...;\ i_n : e_n \bullet p$	existential quantification	there exist ($\exists$) values of i_1 in set e_1, ..., i_n in set e_n ($i_1 : e_1;\ ...;\ i_n : e_n$) such that ($\bullet$) p is *true*
$\forall\ i_1 : e_1;\ ...;\ i_n : e_n \bullet p$	universal quantification	for all ($\forall$) values of i_1 in set e_1, ..., i_n in set e_n ($i_1 : e_1;\ ...;\ i_n : e_n$), it is true that ($\bullet$) p is *true*
$\exists_1\ i_1 : e_1;\ ...;\ i_n : e_n \bullet p$	unique existential quantification	there exists exactly one ($\exists_1$) configuration of values i_1 in set e_1, ..., i_n in set e_n ($i_1 : e_1;\ ...;\ i_n : e_n$) such that ($\bullet$) p is *true*

Certain abbreviations in the writing of quantifications are permitted, as given in Table 5. Some of their expansions involve lesser abbreviations.

Table 5 — Abbreviations in quantifications in metalanguage

Notation	Definition
$i_1, i_2, ..., i_n : e$	$i_1 : e;\ i_2, ..., i_n : e$
$\exists\ i_1, i_2, ..., i_n : e \mid p_1 \bullet p_2$	$\exists\ i_1, i_2, ..., i_n : e \bullet p_1 \wedge p_2$
$\forall\ i_1, i_2, ..., i_n : e \mid p_1 \bullet p_2$	$\forall\ i_1, i_2, ..., i_n : e \bullet (\neg\ p_1) \vee p_2$
$\exists_1\ i_1, i_2, ..., i_n : e \mid p_1 \bullet p_2$	$\exists_1\ i_1, i_2, ..., i_n : e \bullet p_1 \wedge p_2$

4.2.3.4 Conditional expression

The conditional expression allows the choice between two alternative values according to the truth or falsity of a given proposition, as defined in Table 6.

Table 6 — Conditional expression in metalanguage

Notation	Definition
if p then e_1 else e_2	either p is *true* and e_1 is the value, or p is *false* and e_2 is the value

4.2.4 Sets

4.2.4.1 Introduction

The notation used here is based on ZF set theory, as described in for example [1], and the presentation here is guided by the order given there. In that theory there are only sets. Members of sets can be only other sets. The word "element" may be used loosely when referring to set members treated as atomic, without regard to their set nature. If metalanguage operations are applied to inappropriate arguments, they produce unspecified results rather than being undefined.

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

4.2.4.2 The universe

The universe, $\mathbb{U}$, denotes a world of sets, providing semantic values for Z expressions. $\mathbb{U}$ is big enough to contain the set `NAME` from which Z names are drawn, and an infinite set. Also, $\mathbb{U}$ is closed under formation of powersets and products. The formation of a suitable $\mathbb{U}$ comprising models of sets and tuples, as needed to model Z, is well-known in ZF set theory and is assumed in this International Standard. A Z binding is modelled by a ZF set of pairs of `NAME` and value. A Z generic is modelled by a ZF set of pairs (a function) from sets (the generic arguments) to a value (the instantiated generic).

4.2.4.3 Propositions about sets and elements

The simplest propositions about sets are the relationships of membership, non-membership, subset and equality between sets or their elements, as detailed in Table 7.

Table 7 — Propositions about sets in metalanguage

Notation	Name	Definition
$e_1 \in e_1$	membership	*true* iff e_1 is a member of set e_2
$e_1 \notin e_2$	non-membership	$\neg\ e_1 \in e_2$
$e_1 \subseteq e_2$	subset	$\forall\ i : e_1 \bullet i \in e_2$
$e_1 = e_2$	equality	for e_1 and e_2 considered as sets, $e_1 \subseteq e_2 \wedge e_2 \subseteq e_1$

4.2.4.4 Basic set operations

ZF set theory constructs its repertoire of set operations starting with the axiom of empty set, then showing how to build up sets using the axioms of pairing and of union, and how to trim them back with the axiom of subset or separation.

For the purposes of this mathematical metalanguage, the simplest form of set comprehension is defined directly using the axiom of separation in Table 8. The existence of a universal set $\mathbb{U}$ is assumed. Other forms of set comprehension are defined in terms of the simplest form, using rules in which i is any name distinct from those already in use. Also defined in Table 8 are notations for empty set, finite set extensions, unions, intersections and differences.

Table 8 — Basic set operations in metalanguage

Notation	Name	Definition
$\{i : e \mid p\}$	set comprehension	subset of elements i of e such that p, by axiom of separation
$\{i_1 : e_1;\ ...;\ i_n : e_n \bullet e\}$	set comprehension	$\{i : \mathbb{U} \mid \exists\ i_1 : e_1;\ ...;\ i_n : e_n \bullet i = e\}$
$\{i_1 : e_1;\ ...;\ i_n : e_n \mid p \bullet e\}$	set comprehension	$\{i : \mathbb{U} \mid \exists\ i_1 : e_1;\ ...;\ i_n : e_n \mid p \bullet i = e\}$
$\varnothing$	empty set	$\{i : \mathbb{U} \mid \mathit{false}\}$
$\{\}$	empty set	$\{i : \mathbb{U} \mid \mathit{false}\}$
$\{e\}$	singleton set	$\{i : \mathbb{U} \mid i = e\}$
$e_1 \cup e_2$	union	$\{i : \mathbb{U} \mid i \in e_1 \vee i \in e_2\}$
$\{e_1, e_2, ..., e_n\}$	set extension	$\{e_1\} \cup \{e_2, ..., e_n\}$
$e_1 \cap e_2$	intersection	$\{i : e_1 \mid i \in e_2\}$
$e_1 \setminus e_2$	difference	$\{i : e_1 \mid i \notin e_2\}$

4.2.4.5 Powersets

The axiom of powers asserts the existence of a powerset, which is the set of all subsets of a set. The set of all finite subsets is a subset of the powerset. It is the smallest set containing the empty set and all singleton subsets of e and closed under the operation of forming the union with singleton subsets of e. Their forms are given in Table 9.

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

Table 9 — Powerset in metalanguage

Notation	Name	Definition
$\mathbb{P}\ e$	set of all subsets	$\{i : \mathbb{U} \mid i \subseteq e\}$
$\mathbb{F}\ e$	set of all finite subsets	$\{i_1 : \mathbb{U} \mid \forall\, i_2 : \mathbb{P}\,\mathbb{P}\, e \mid \varnothing \in i_2 \wedge (\forall\, i_3 : i_2 \bullet \forall\, i_4 : e \bullet i_3 \cup \{i_4\} \in i_2) \bullet i_1 \in i_2\}$

4.2.5 Natural numbers

Natural numbers are not primitive in ZF set theory, but there are several well established ways of representing them. The choice of coding is irrelevant here and so is not specified. There are notations to measure the cardinality of a finite set, to define addition of natural numbers and to form the set of natural numbers between two stated natural numbers, as given in Table 10.

Table 10 — Operations on natural numbers in metalanguage

Notation	Definition
$e_1 + e_2$	sum of natural numbers e_1 and e_2
$\#\ e$	cardinality of finite set e
$e_1 \mathinner{..} e_2$	set of natural numbers between e_1 and e_2 inclusive

4.2.6 Names

Names are needed for this International Standard. There are several ways of representing names in ZF set theory. The choice of coding is irrelevant here and so is not specified. Only one operation is needed on names; it is an infix operation with highest precedence, and is defined in Table 11.

Table 11 — Decorations of names in metalanguage

Notation	Definition
$i\ decor\ ^{+}$	the name that is like i but with the extra stroke $^{+}$

4.2.7 Tuples and Cartesian products

Tuples and Cartesian products are not primitive in ZF set theory, but there are various ways in which they may be represented within that theory, such as the well-known encoding given by Kuratowski [1]. The choice of coding is irrelevant here and so is not specified. In this International Standard, the syntactic context of the metalanguage expression denoting a value always determines whether or not that value is to be interpreted as an encoding of a tuple. Therefore there is never any possibility of accidental confusion between the encoding used to represent the tuple and any other value that is not a tuple.

In this mathematical metalanguage, tuples and Cartesian products with more than two components are interpreted as nested binary tuples and products, unlike in Z.

The syntactic forms are given in Table 12.

The brackets delimiting a pair or tuple extension written with commas shall not be omitted—they are not grouping parentheses.

4.2.8 Function comprehensions

Table 13 defines the notation for λ, which is a form of comprehension convenient when defining functions.

4.2.9 Relations

A relation is defined to be a set of pairs. There are several operations involving relations, which are given equivalences in Table 14. A proposition about relations is given in Table 15.

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

Table 12 — Tuples and Cartesian products in metalanguage

Notation	Name	Definition
(e_1, e_2)	pair	
$e_1 \mapsto e_2$	maplet	(e_1, e_2)
$first\ e$		$first(e_1, e_2) = e_1$
$second\ e$		$second(e_1, e_2) = e_2$
$e_1 \times e_2$		$\{i_1 : e_1;\ i_2 : e_2 \bullet (i_1, i_2)\}$
$(e_1, e_2, ..., e_n)$	tuple extension	$(e_1, (e_2, ..., e_n))$ where $n > 2$
$e_1 \times e_2 \times ... \times e_n$	Cartesian product	$e_1 \times (e_2 \times ... \times e_n)$ where $n > 2$
$e \uparrow 1$		e
$e \uparrow n$	iterated product	$e \times ... \times e$ where there are $n \geq 2$ occurrences of e

Table 13 — Function comprehensions in metalanguage

Notation	Definition
$\lambda\ i : e_1 \bullet e_2$	$\{i : e_1 \bullet i \mapsto e_2\}$
$\lambda\ i : e_1 \mid p \bullet e_2$	$\{i : e_1 \mid p \bullet i \mapsto e_2\}$
$\lambda\ i_1 : e_1;\ ...;\ i_n : e_n \bullet e$	$\{i_1 : e_1;\ ...;\ i_n : e_n \bullet (i_1, ..., i_n) \mapsto e\}$
$\lambda\ i_1 : e_1;\ ...;\ i_n : e_n \mid p \bullet e$	$\{i_1 : e_1;\ ...;\ i_n : e_n \mid p \bullet (i_1, ..., i_n) \mapsto e\}$

Table 14 — Relations in metalanguage

Notation	Name	Definition
$id\ e$	identity function	$\lambda\ i : e \bullet i$
$dom\ e$	domain	$\{i : e \bullet first\ i\}$
$e_1 \lhd e_2$	domain restriction	$\{i : e_2 \mid first\ i \in e_1\}$
$e_1 \ntriangleleft e_2$	domain subtraction	$\{i : e_2 \mid first\ i \notin e_1\}$
$e_1 (\!\mid e_2 \mid\!)$	relational image	$\{i : e_1 \mid first\ i \in e_2 \bullet second\ i\}$
$e_1 \fatsemi e_2$	relational composition	$\{i_1 : e_1;\ i_2 : e_2 \mid second\ i_1 = first\ i_2 \bullet first\ i_1 \mapsto second\ i_2\}$
$e_1 \oplus e_2$	relational overriding	$((dom\ e_2) \ntriangleleft e_1) \cup e_2$

Table 15 — Proposition about relations in metalanguage

Notation	Name	Definition
$e_1 \approx e_2$	compatible relations	$(dom\ e_2) \lhd e_1 = (dom\ e_1) \lhd e_2$

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

4.2.10 Functions

A function is a particular form of relation, where each domain element has only one corresponding range element. Table 16 shows the various forms of function that are identified, each being a set of functions.

Table 16 — Functions in metalanguage

Notation	Name	Definition
$e_1 \nrightarrow e_2$	functions	$\{i_1 : \mathbb{P}\,(e_1 \times e_2) \mid \forall\ i_2, i_3 : i_1 \mid \mathit{first}\ i_2 = \mathit{first}\ i_3 \bullet \mathit{second}\ i_2 = \mathit{second}\ i_3\}$
$e_1 \rightarrow e_2$	total functions	$\{i : e_1 \nrightarrow e_2 \mid \mathit{dom}\ i = e_1\}$
$e_1 \nrightarrow\!\!\!\!\rightarrow e_2$	finite functions	$\{i : \mathbb{F}\,(e_1 \times e_2) \mid i \in e_1 \nrightarrow e_2\}$

4.2.10.1 Application

A function can be juxtaposed with an argument to produce a result, using the notation of Table 17. Metalanguage notations introduced above that match the $e_1\ e_2$ pattern, such as $\mathit{dom}\ e$, are not applications in this sense.

Table 17 — Application in metalanguage

Notation	Name	Definition
$e_1\ e_2$	application	if there exists a unique e_3 such that $e_2 \mapsto e_3$ is in e_1, then the value of $e_1\ e_2$ is e_3, otherwise each $e_1\ e_2$ has a fixed but unknown value

4.2.11 Sequences

A sequence is a particular form of function, where the domain elements are all the natural numbers from 1 to the length of the sequence.

Table 18 — Sequences in metalanguage

Notation	Name	Definition
$\langle e_1, ..., e_n \rangle$	sequence	$\{1 \mapsto e_1, ..., n \mapsto e_n\}$

4.2.12 Disjointness

A labelled family of sets is disjoint when any distinct pair yields sets with no members in common.

Table 19 — Disjointness in metalanguage

Notation	Name	Definition
$\mathit{disjoint}\ e$	disjointness	$\forall\, e_1, e_2 : \mathit{dom}\ e \mid e_1 \neq e_2 \bullet e\ e_1 \cap e\ e_2 = \varnothing$

4.3 Transformation metalanguage

Transformation rules map parse trees of phrases to other parse trees. Each transformation rule is written in the following form.

$$\mathit{concrete\ phrase\ template} \quad \Longrightarrow \quad \mathit{less\ concrete\ phrase\ template}$$

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

EXAMPLE 1 The syntactic transformation rule for a schema definition paragraph, and an informal reading of it, are as follows.

$$\texttt{SCH}\ i\ t\ \texttt{END} \quad \Longrightarrow \quad \texttt{AX}\ [i == t]\ \texttt{END}$$

A schema definition paragraph is formed from a box token `SCH`, a name i, a schema text t, and an `END` token. An equivalent axiomatic description paragraph is that which would be written textually as a box token `AX`, a [token, the original name i, a == token, the original schema text t, a] token, and an `END` token.

EXAMPLE 2 The semantic transformation rule for a schema hiding expression, and an informal reading of it, are as follows.

$$(e \ ⨾\ \mathbb{P}[\sigma]) \setminus (i_1, ..., i_n) \quad \Longrightarrow \quad \exists\ i_1 : carrier\ (\sigma\ i_1);\ ...;\ i_n : carrier\ (\sigma\ i_n) \bullet e$$

A schema with signature σ from which some names are hidden is semantically equivalent to the schema existential quantification of the hidden names from the schema. Each name is declared with the set that is the carrier set of the type of the name in the signature of the schema.

The phrase templates are patterns; they are not specific sentences and they are not written in the syntactic metalanguage. These patterns are written in a notation based on the concrete and annotated syntaxes, with metavariables appearing in place of syntactically well-formed phrases. The metavariables are defined in Tables 20 and 21 (the phrases being defined in clauses 7 and 8, and the operator words in 7.4.4). Where several phrases of the same syntactic classes have to be distinguished, these metavariables are given distinct numeric subscripts. The letters $_k$, $_m$, $_n$, $_r$ are used as metavariables for such numeric subscripts. The patterns can be viewed either as using the non-terminal symbols of the Z lexis with the -tok suffixes omitted from mathematical symbols, or as using the mathematical rendering with the box tokens in place of paragraph outlines.

Table 20 — Metavariables for phrases

Symbol	Definition
a	denotes an `ExpressionList` phrase (a for list argument).
b	denotes a list of digits within a `NUMERAL` token.
c	denotes a digit within a `NUMERAL` token.
D	denotes a `Paragraph` phrase.
d	denotes a `Declaration` phrase.
e	denotes an `Expression` phrase.
f	denotes a free type's `NAME` token.
g	denotes an injection's `NAME` token (g for in$g$ection).
h	denotes an element's `NAME` token (h for $h$element).
i, j	denote `NAME` tokens or `DeclName` or `RefName` phrases (i for identifier).
p	denotes a `Predicate` phrase.
s	denotes a `Section` phrase.
t	denotes a `SchemaText` phrase (t for text).
u, v, w, x, y	denote distinct names for new local declarations.
z	denotes a `Specification` sentence.
τ	denotes a `Type` phrase.
σ	denotes a `Signature` phrase.
$+$	denotes a `STROKE` token.
$*$	denotes a `{ STROKE }` phrase.
...	denotes elision of repetitions of surrounding phrases, the total number of repetitions depending on syntax.

The applicability of a transformation rule can be guarded by a condition written above the $\Longrightarrow$ symbol. Local definitions can be associated with a transformation rule by appending a *where* clause, in which later definitions can refer to earlier definitions.

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

Table 21 — Metavariables for operator words

Symbol	Definition
el	denotes an `EL` token.
elp	denotes an `ELP` token.
er	denotes an `ER` token.
ere	denotes an `ERE` token.
erep	denotes an `EREP` token.
erp	denotes an `ERP` token.
es	denotes an `ES` token.
ess	denotes an `ES` token or `SS` token.
in	denotes an `I` token.
ip	denotes an `IP` token or $\in$ token or $=$ token.
ln	denotes an `L` token.
lp	denotes an `LP` token.
post	denotes a `POST` token.
postp	denotes a `POSTP` token.
pre	denotes a `PRE` token.
prep	denotes a `PREP` token.
sr	denotes an `SR` token.
sre	denotes an `SRE` token.
srep	denotes an `SREP` token.
srp	denotes an `SRP` token.
ss	denotes an `SS` token.

The transformation rule metalanguage is used in defining characterisation rules, syntactic transformation rules, type inference rules, and semantic transformation rules (clauses 9, 12, 13 and 14).

4.4 Type inference rule metalanguage

Each type inference rule is written in the following form.

$$\frac{\textit{type subsequents}}{\textit{type sequent}}(\textit{side-condition})$$

where local-declaration
and ...

This can be read as: if the *type subsequents* are valid, and the *side-condition* is true, then the *type sequent* is valid, in the context of the zero-or-more *local-declarations*. The side-condition is optional; if omitted, the type inference rule is equivalent to one with a true side-condition.

The annotated syntax (see 10.2) establishes notation for writing types as `Type` phrases and for writing signatures as `Signature` phrases. The $\circ$ operator allows annotations such as types to be associated with other phrases. Determining whether a type sequent is valid or not involves manipulation of types and signatures. This requires viewing types and signatures as values, and having a mathematical notation to do the manipulation. Signatures are viewed as functions from names to type values. `Type` is used to denote the set of type values as well as the set of type phrases, the appropriate interpretation being distinguished by context of use. Similarly, `NAME` is also used to denote a set of name values. These values all lie within the type universe. A type's `NAME` has a corresponding type value in the type universe whereas a type's carrier set is in the semantic universe.

Type values are formed from just finite sets and ordered pairs, so the mathematical metalanguage introduced in 4.2 suffices for their manipulation.

Details of which names are in scope are kept in environments. The various kinds of environment are defined in

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

Table 22, and metavariables for environments are defined in Table 23.

Table 22 — Environments

Symbol	Definition
`TypeEnv`	denotes type environments, where `TypeEnv` == `NAME` ⇸ `Type`. Type environments associate names with types. They are like signatures, but are used in different contexts.
`SectTypeEnv`	denotes section-type environments, where `SectTypeEnv` == `NAME` ⇸ (`NAME` × `Type`). Section-type environments associate names of declarations with the name of the ancestral section that originally declared the name paired with its type.
`SectEnv`	denotes section environments, where `SectEnv` == `NAME` ⇸ `SectTypeEnv`. Section environments associate section names with section-type environments.

Table 23 — Metavariables for environments

Symbol	Definition
Σ	denotes a type environment, Σ : `TypeEnv`.
Γ	denotes a section-type environment, Γ : `SectTypeEnv`.
Λ	denotes a section environment, Λ : `SectEnv`.

Variables over the type universe are defined in Table 24, and a relation on types is defined in Table 25.

Table 24 — Variables over type universe

Symbol	Definition
α	denotes a type, α : `Type`.
β	denotes a signature, β : `Signature`, which may be a type environment.
γ	denotes a section-type environment, γ : `SectTypeEnv`.
δ	denotes a section environment, δ : `SectEnv`.

Table 25 — Type relations

Symbol	Definition
generic_type τ	asserts that τ is a generic type.

Type sequents are written using a $\vdash$ symbol superscripted with a mnemonic letter to distinguish the syntax of the phrase appearing to its right (Table 26).

NOTE 1 These superscripts are the same as the superscripts used on the ⟦ ⟧ semantic brackets in the semantic relations below (Table 29).

The annotated phrases to the right of $\vdash$ in type sequents are phrase templates written using the same metavariables as the syntactic transformation rules (Table 20).

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

4 Metalanguages

Table 26 — Type sequents

Symbol	Definition
$\vdash^{\mathcal{Z}} z$	a type sequent asserting that specification z is well-typed.
$\Lambda \vdash^{\mathcal{S}} s \mathrel{:\!\!:} \Gamma$	a type sequent asserting that, in the context of section environment Λ, section s has section-type environment Γ.
$\Sigma \vdash^{\mathcal{D}} D \mathrel{:\!\!:} \sigma$	a type sequent asserting that, in the context of type environment Σ, the paragraph D has signature σ.
$\Sigma \vdash^{\mathcal{P}} p$	a type sequent asserting that, in the context of type environment Σ, the predicate p is well-typed.
$\Sigma \vdash^{\mathcal{E}} e \mathrel{:\!\!:} \tau$	a type sequent asserting that, in the context of type environment Σ, the expression e has type τ.

EXAMPLE The type inference rule for a schema conjunction expression, and its informal reading, are as follows.

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \mathrel{:\!\!:} \tau_1 \qquad \Sigma \vdash^{\mathcal{E}} e_2 \mathrel{:\!\!:} \tau_2}{\Sigma \vdash^{\mathcal{E}} (e_1 \mathrel{:\!\!:} \tau_1) \wedge (e_2 \mathrel{:\!\!:} \tau_2) \mathrel{:\!\!:} \tau_3} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta_1] \\ \tau_2 = \mathbb{P}[\beta_2] \\ \beta_1 \approx \beta_2 \\ \tau_3 = \mathbb{P}[\beta_1 \cup \beta_2] \end{array} \right)$$

In a schema conjunction expression $e_1 \wedge e_2$, expressions e_1 and e_2 shall be schemas, and their signatures shall be compatible. The type of the whole expression is that of the schema whose signature is the union of those of expressions e_1 and e_2.

NOTE 2 The metavariables β_1 and β_2 in this example denote syntactic phrases. These are mapped implicitly to type values, so that the set union can be computed, and the resulting signature is implicitly mapped back to a syntactic phrase. These mappings are not made explicit as they would make the type inference rules harder to read, e.g. $[\![\, [\![\beta_1]\!] \cup [\![\beta_2]\!] \,]\!]$.

This metalanguage is used in defining type inference rules (clause 13).

4.5 Semantic relation metalanguage

Most semantic relations are equations written in the following form.

$$[\![\mathit{phrase\ template}]\!] \quad = \quad \mathit{semantics}$$

Where the definition is only partial, the equality notation is not appropriate, and instead a lower bound is specified on the semantics.

$$\mathit{semantics} \quad \subseteq \quad [\![\; \mu \; e_1 \bullet e_2 \;]\!]^{\mathcal{E}}$$

The phrase templates use the same metavariables as used by the syntactic transformation rules (Table 20).

Symbols concerned with the domains of the semantic definitions are listed in Tables 27 and 28.

The meaning of a phrase template is given by a semantic relation from the Z phrase in terms of operations of ZF set theory on the semantic universe. There are different semantic relations for each syntactic notation, written using the conventional $[\![\;]\!]$ semantic brackets, but here superscripted with a mnemonic letter to distinguish the syntax of phrase appearing within them (Table 29).

NOTE The superscripts are the same as those used on the $\vdash$ of type sequents in the type inference rules above (Table 26).

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

Table 27 — Semantic universe

Symbol	Definition
$\mathbb{W}$	denotes the world of semantic values for Z expressions that are not generic, where $\mathbb{W} : \mathbb{P}\,\mathbb{U}$.
$Model$	denotes models, where $Model == \mathtt{NAME} \twoheadrightarrow \mathbb{U}$. Models associate names of declarations with semantic values. They are applied only to names in their domains, as guaranteed by well-typedness.
$SectionModels$	denotes functions from sections' names to their sets of models, where $SectionModels == \mathtt{NAME} \twoheadrightarrow \mathbb{P}\,Model$.

Table 28 — Variables over semantic universe

Symbol	Definition
M	denotes a model, $M : Model$.
T	denotes a section's name and its set of models, $T : SectionModels$.
t	denotes a binding semantic value, $t : \mathtt{NAME} \twoheadrightarrow \mathbb{W}$.
u	denotes a generic semantic value, $u : \mathbb{U} \setminus \mathbb{W}$.
w, x, y	denote non-generic semantic values, $w : \mathbb{W}$; $x : \mathbb{W}$; $y : \mathbb{W}$.

Table 29 — Semantic relations

Symbol	Definition
$[\![\, z \,]\!]^{\mathcal{Z}}$	denotes the meaning of specification z, where $[\![\, z \,]\!]^{\mathcal{Z}} \in SectionModels$. The meaning of a specification is the function from its sections' names to their sets of models.
$[\![\, s \,]\!]^{\mathcal{S}}$	denotes the meaning of section s, where $[\![\, s \,]\!]^{\mathcal{S}} \in SectionModels \nrightarrow SectionModels$. The meaning of a section is given by the extension of a $SectionModels$ function with an extra maplet corresponding to the given section.
$[\![\, D \,]\!]^{\mathcal{D}}$	denotes the meaning of paragraph D, where $[\![\, D \,]\!]^{\mathcal{D}} \in Model \leftrightarrow Model$. The meaning of a paragraph relates a model to that model extended according to that paragraph.
$[\![\, p \,]\!]^{\mathcal{P}}$	denotes the meaning of predicate p, where $[\![\, p \,]\!]^{\mathcal{P}} \in \mathbb{P}\,Model$. The meaning of a predicate is the set of all models in which that predicate is true.
$[\![\, e \,]\!]^{\mathcal{E}}$	denotes the meaning of expression e, where $[\![\, e \,]\!]^{\mathcal{E}} \in Model \rightarrow \mathbb{W}$. The meaning of an expression is a function returning the semantic value of the expression in the given model.
$[\![\, \tau \,]\!]^{\mathcal{T}}$	denotes the meaning of type τ, where $[\![\, \tau \,]\!]^{\mathcal{T}} \in Model \nrightarrow \mathbb{P}\,\mathbb{U}$. The meaning of a type is the semantic value of its carrier set, as determined from the given model.

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

EXAMPLE The semantic relation for a conjunction predicate, and its informal reading, are as follows. The conjunction predicate $p_1 \wedge p_2$ is *true* if and only if p_1 and p_2 are *true*.

$$[\![p_1 \wedge p_2]\!]^{\mathcal{P}} \quad = \quad [\![p_1]\!]^{\mathcal{P}} \cap [\![p_2]\!]^{\mathcal{P}}$$

In terms of the semantic universe, it is *true* in those models in which both p_1 and p_2 are *true*, and is *false* otherwise.

Within the semantic relations, the idioms listed in Table 30 occur frequently.

Table 30 — Semantic idioms

Idiom	Description
$[\![e]\!]^{\mathcal{E}} M$	denotes the value of expression e in model M
$M \oplus t$	denotes the model M giving semantic values for more global declarations overridden by the binding t giving the semantic values of locally declared names

Semantic relation metalanguage is used in defining semantic relations (clause 15).

5 Conformance

5.1 Phases of the definition

The definition of the Z notation is divided into a sequence of phases, as illustrated in Figure 1. Each arrow represents a phase from a representation of a Z specification at its source to another representation of the Z specification at its target. The phase is named at the left margin. Some phases detect errors in the specification; these are shown drawn off to the right-hand side.

NOTE 1 Figure 1 shows the order in which the phases are applied, and where errors are detected; it does not show information flows.

NOTE 2 The arrows are analogous to total and partial function arrows in the Z mathematical toolkit, but drawn vertically.

5.2 Conformance requirements

5.2.1 Specification conformance

For a Z specification to conform to this International Standard, no errors shall be detected by any of the phases shown in Figure 1. In words, for a Z specification to conform to this International Standard, its formal text shall be valid mark-up of a sequence of Z characters, that can be lexed as a valid sequence of tokens, that can be parsed as a sentence of the concrete syntax, and that is well-typed according to the type inference system.

NOTE The presence of sections that have no models does not affect the conformance of their specification.

5.2.2 Mark-up conformance

A mark-up for Z based on LaTeX [10] conforms to this International Standard if and only if it follows the rules given for LaTeX mark-up in A.2.

A mark-up for Z used in e-mail correspondence conforms to this International Standard if and only if it follows the rules given for e-mail mark-up in A.3.

Mark-up for Z based on any other mark-up language is permitted; it shall be possible to define a functional mapping from that mark-up to sequences of Z characters.

The ISO/IEC 10646 [5][6] representation may be used directly.

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

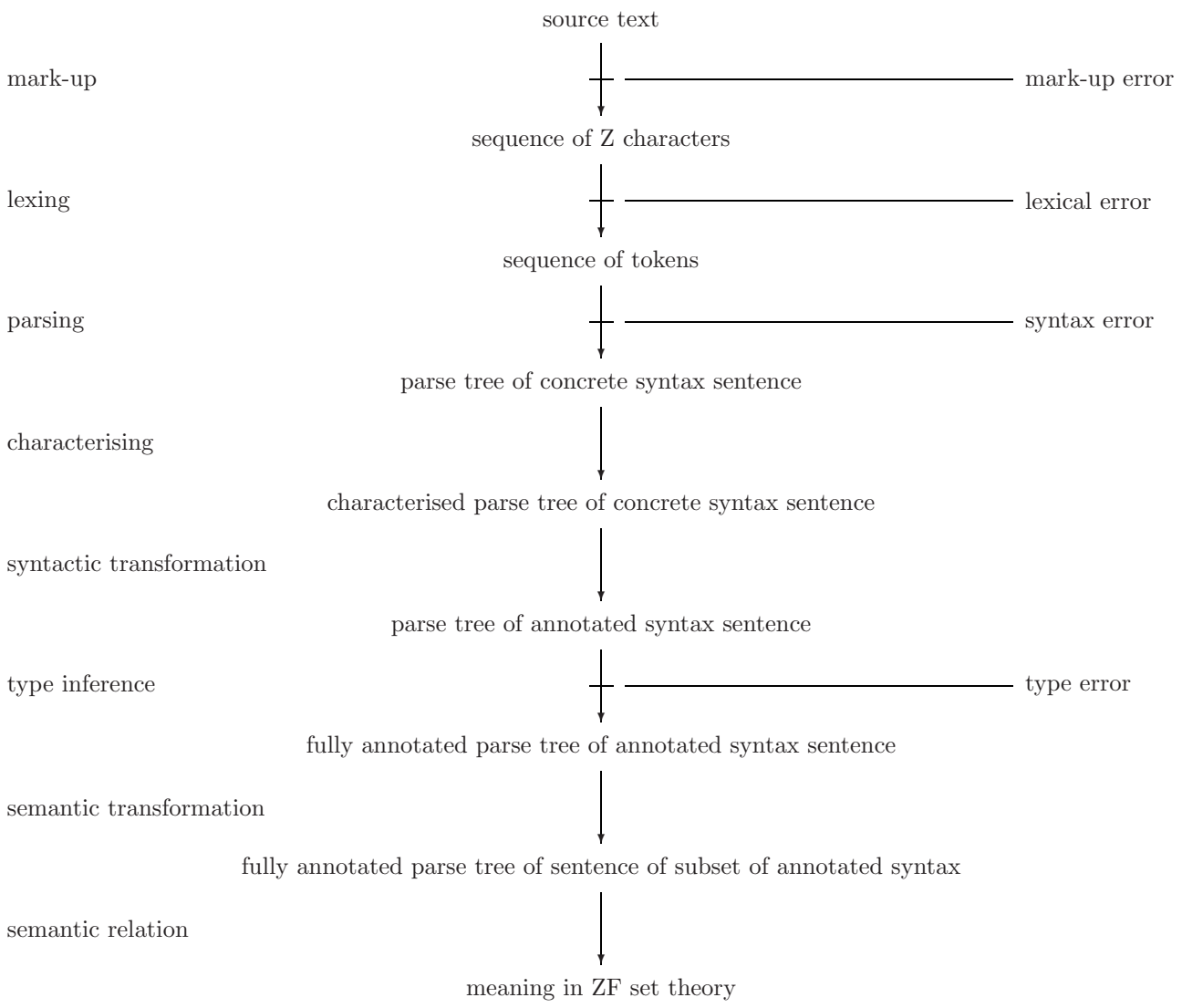

Figure 1 — Phases of the definition

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

5 Conformance

5.2.3 Deductive system conformance

A Z deductive system conforms to this International Standard if and only if its rules are sound with respect to the semantics, i.e. if both of the following conditions hold:

a) all of its axioms hold in all models of all Z specifications, i.e. for any axiom p,

$$[\![p]\!]^{\mathcal{P}} = Model$$

b) all of its rules of inference have the property that the intersection of the sets of models of each of the premises is contained in the model of the conclusion, i.e. for any rule of inference where p is deduced from p_1, ... p_n,

$$[\![p_1]\!]^{\mathcal{P}} \cap \ldots \cap [\![p_n]\!]^{\mathcal{P}} \subseteq [\![p]\!]^{\mathcal{P}}$$

All constraints appearing before a conjecture in a specification may be used as premises in inferences about that conjecture. The documentation of a Z deductive system should clarify whether or not it allows constraints appearing after a conjecture in a specification to be used as premises in inferences about that conjecture.

The semantic relations assigning meanings to Z phrases are defined loosely by this International Standard, so that there is a set of possible conforming semantic relations. A deductive system conforms if and only if its rules are sound with respect to one or more of these semantic relations. The documentation of a Z deductive system should indicate the semantic relations to which its rules are sound.

5.2.4 Mathematical toolkit conformance

A Z section whose name is the same as a section of the mathematical toolkit of annex B conforms to this International Standard if and only if it defines the same set of models as that section of the mathematical toolkit. A Z section whose name is *prelude* conforms to this International Standard if and only if it defines the same set of models as the section in clause 11.

A mathematical toolkit conforms to this standard if it defines a conformant section called standard_toolkit.

NOTE 1 The set of models defined by a section within a specification may be found by applying the meaning of the specification to the section's name.

NOTE 2 A conforming section of the toolkit may formulate its definitions differently from those in annex B.

NOTE 3 A conforming section of the toolkit may partition its definitions amongst parent sections that differ from those in annex B.

NOTE 4 Alternative and additional toolkits are not precluded, but are required to have different section names to avoid confusion.

NOTE 5 Some names are loosely defined, such as $\mathbb{A}$, and may be further constrained by sections that use toolkit sections, but not by toolkit sections themselves.

5.2.5 Support tool conformance

A strongly conforming Z support tool shall recognise at least one conforming mark-up, accepting all conforming Z specifications presented to it, and rejecting all non-conforming Z specifications presented to it. A weakly conforming Z support tool shall never accept a non-conforming Z specification, nor reject a conforming Z specification, but it may state that it is unable to determine whether or not a Z specification conforms.

NOTE Strong conformance can be summarised as always being right, whereas weak conformance is never being wrong.

EXAMPLE A tool would be weakly conformant if it were to announce its inability to determine the conformance of a Z specification that used names longer than the tool could handle, but would be non-conformant if it silently truncated long names.

Certain exceptions to general rules are anticipated and permitted in subsequent clauses, because of, for example, implementation considerations or for backwards compatibility with pre-existing tools.

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

5.3 Structure of this document

The phases in the definition of the Z notation, and the representations of specifications manipulated by those phases, as illustrated in Figure 1, are specified in the following clauses and annexes.

Annex A, Mark-ups, specifies two source text representations and corresponding mark-up phases for translating source text to sequences of Z characters.

Clause 6 specifies the Z characters by their appearances and their names in ISO/IEC 10646.

Clause 7, Lexis, specifies tokens and the lexing phase that translates a sequence of Z characters to a sequence of tokens.

Clause 8 specifies the grammar of the concrete syntax, and hence abstractly specifies the parsing phase that translates a sequence of tokens to a parse tree of a concrete syntax sentence.

Clause 9 specifies the characterising phase, during which characteristic tuples are made explicit in the parse tree of a concrete syntax sentence.

Clause 10 specifies the grammar of the annotated syntax, defining the target language for the syntactic transformation phase.

Clause 11 specifies the prelude section, providing the initial environment of definitions.

Clause 12 specifies the syntactic transformation phase that translates a parse tree of a concrete syntax sentence to a parse tree of an equivalent annotated syntax sentence.

Clause 13 specifies the type inference phase, during which type annotations are added to the parse tree of the annotated syntax sentence, and reference expressions that refer to generic definitions are translated to generic instantiation expressions.

Clause 14 specifies the semantic transformation phase, during which some annotated parse trees are translated to equivalent other annotated parse trees.

Clause 15 specifies the semantic relation between a sentence of the remaining annotated syntax and its meaning in ZF set theory.

Annex C duplicates those parts of the definition that fit into an organisation by concrete syntax production.

6 Z characters

6.1 Introduction

A Z character is the smallest unit of information in Z. Z characters are used to build tokens (clause 7), which are in turn the units of information in the concrete syntax (clause 8). The Z characters are defined by reference to ISO/IEC 10646-1 [5] and ISO/IEC 10646-2 [6]: the appearance, code position and name of each Z character are listed.

Many Z characters are not present in the standard 7-bit ASCII encoding [4]. It is possible to represent Z characters in ASCII, by defining a mark-up, where several ASCII characters are used together to represent a single Z character. This International Standard defines some ASCII mark-ups in annex A by relation to the ISO/IEC 10646 representation (henceforth called UCS) defined here. Other mark-ups of Z characters can similarly be defined by relation to the UCS representation.

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

6 Z characters

6.2 Formal definition of Z characters

Copyright notice

The reproduction of this clause is permitted on the understanding that this material is public domain, and on the condition that this International Standard is referenced as the source document. With the exception of clauses 6.2, 7.2, 8.2, 11 and annex B, all other parts of the text are subject to the usual copyright rules stated on page ii of this International Standard.

```
ZCHAR          = DIGIT | LETTER | SPECIAL | SYMBOL ;

DIGIT          = DECIMAL
               | ? other UCS chars with Number property but Number, Decimal Digit (as supported) ?
               ;

DECIMAL        = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
               | ? any other UCS characters with Number, Decimal Digit property (as supported) ?
               ;

LETTER         = LATIN | GREEK | OTHERLETTER
               | ? any characters of the mathematical toolkit with letter property (as supported) ?
               | ? any other UCS characters with letter property (as supported) ?
               ;

LATIN          = '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          = 'Δ' | 'Ξ' | 'θ' | 'λ' | 'μ' ;

OTHERLETTER    = '𝔸' | 'ℕ' | 'ℙ' ;

SPECIAL        = STROKECHAR | WORDGLUE | BRACKET | BOXCHAR | NLCHAR | SPACE ;

STROKECHAR     = '′' | '!' | '?' ;

WORDGLUE       = '↗' | '↙' | '↘' | '↖' | '_' ;

BRACKET        = '(' | ')' | '[' | ']' | '{' | '}' | '⦇' | '⦈' | '⟪' | '⟫' ;

BOXCHAR        = ZEDCHAR | AXCHAR | SCHCHAR | GENCHAR | ENDCHAR ;

SYMBOL         = '|' | '&' | '⊢' | '∧' | '∨' | '⇒' | '⇔' | '¬' | '∀' | '∃'
               | '×' | '/' | '=' | '∈' | ':' | ';' | ',' | '.' | '•'
               | '\' | '↾' | '⨾' | '≫' | '+'
               | ? any characters of the mathematical toolkit with neither letter or
                   number property (as supported) ?
               | ? any other UCS characters with neither letter or
                   number property and that are not in SPECIAL(as supported) ?
               ;
```


Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

6.3 Additional restrictions and notes

The word *supported* means available for use in presenting a specification.

The characters enumerated in the formal definition are those used by the Z core language; they shall be supported. If the mathematical toolkit is supported, then its characters shall be supported. The "other UCS characters" may also be supported, extending `DIGIT`, `DECIMAL`, `LETTER` or `SYMBOL` according to their property, but not extending `SPECIAL`. Use of characters that are absent from UCS is permitted, but there is no standard way of distinguishing which of `DIGIT`, `DECIMAL`, `LETTER` or `SYMBOL` (not `SPECIAL`) they extend, and specifications using them might not be interchangeable between tools.

NOTE 1 `SPACE` is a Z character that serves to separate two sequences of Z characters that would otherwise be mis-lexed as a single token.

NOTE 2 `BOXCHAR` characters correspond to Z's distinctive boxes around paragraphs. The `NLCHAR` character is used to mark a hard newline (see 7.5).

6.4 Z character representations

6.4.1 Introduction

The following tables show the Z characters in their mathematical representation. (Other representations are given in annex A.) The columns give:

Math: The representation for rendering the character on a high resolution device, such as a bit-mapped screen, or on paper (either hand-written, or printed).

Code position: The encoding of the Z character in UCS.

Character name: The name for the character in UCS.

NOTE The code position column is included to assist location of the characters by users; it is not necessary for definition of the characters.

6.4.2 Decimal digit characters

Math	Code position	Character name
0	0000 0030	DIGIT ZERO
⋮	⋮	⋮
9	0000 0039	DIGIT NINE

For each `DECIMAL` character, UCS defines a corresponding *decimal digit value*. This is used in 7.3.

6.4.3 Letter characters

6.4.3.1 Latin alphabet characters

Math	Code position	Character name
A	0000 0041	LATIN CAPITAL LETTER A
⋮	⋮	⋮
Z	0000 005A	LATIN CAPITAL LETTER Z
a	0000 0061	LATIN SMALL LETTER A
⋮	⋮	⋮
z	0000 007A	LATIN SMALL LETTER Z

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

6 Z characters

6.4.3.2 Greek alphabet characters

The Greek alphabet characters used by the Z core language are those listed here.

Math	Code position	Character name
Δ	0000 0394	GREEK CAPITAL LETTER DELTA
Ξ	0000 039E	GREEK CAPITAL LETTER XI
θ	0000 03B8	GREEK SMALL LETTER THETA
λ	0000 03BB	GREEK SMALL LETTER LAMBDA
μ	0000 03BC	GREEK SMALL LETTER MU

6.4.3.3 Other Z core language letter characters

The other Z core language characters with UCS *letter* property are those of the prelude (clause 11), as listed here.

Math	Code position	Character name
$\mathbb{A}$	0001 D538	MATHEMATICAL DOUBLE-STRUCK CAPITAL A
$\mathbb{N}$	0000 2115	DOUBLE-STRUCK CAPITAL N
$\mathbb{P}$	0000 2119	DOUBLE-STRUCK CAPITAL P

6.4.4 Special characters

6.4.4.1 Stroke characters

Math	Code position	Character name
$'$	0000 02B9	MODIFIER LETTER PRIME
!	0000 0021	EXCLAMATION MARK
?	0000 003F	QUESTION MARK

6.4.4.2 Word glue characters

The characters '$\nearrow$', '$\swarrow$', '$\searrow$', and '$\nwarrow$' may be presented as in-line literals, or they may indicate a raising/lowering of the text, and possible size change. Such rendering details are not defined here.

Math	Code position	Character name
$\nearrow$	0000 2197	NORTH EAST ARROW
$\swarrow$	0000 2199	SOUTH WEST ARROW
$\searrow$	0000 2198	SOUTH EAST ARROW
$\nwarrow$	0000 2196	NORTH WEST ARROW
_	0000 005F	LOW LINE

6.4.4.3 Bracket characters

Math	Code position	Character name
(	0000 0028	LEFT PARENTHESIS
)	0000 0029	RIGHT PARENTHESIS
[	0000 005B	LEFT SQUARE BRACKET
]	0000 005D	RIGHT SQUARE BRACKET
{	0000 007B	LEFT CURLY BRACKET
}	0000 007D	RIGHT CURLY BRACKET
$\langle\!\vert$	0000 2989	Z NOTATION LEFT BINDING BRACKET
$\vert\!\rangle$	0000 298A	Z NOTATION RIGHT BINDING BRACKET
$\langle\langle$	0000 300A	LEFT DOUBLE ANGLE BRACKET
$\rangle\rangle$	0000 300B	RIGHT DOUBLE ANGLE BRACKET

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

6.4.4.4 Box characters

Box characters are assembled to form box tokens (see 7.2), which in turn correspond to boxes around paragraphs (see 8.5). The simple renderings of the characters are suggestive of the boxes around paragraphs. The `ENDCHAR` character is used to mark the end of a `Paragraph`.

NOTE These box characters are not intended to be used for rendering the boxes around paragraphs.

Z character	Simple rendering	Code position	Character name
`ZEDCHAR`	⎸	0000 2028	LINE SEPARATOR
`AXCHAR`	⎸	0000 2577	BOX DRAWINGS LIGHT DOWN
`SCHCHAR`	⌜	0000 250C	BOX DRAWINGS LIGHT DOWN AND RIGHT
`GENCHAR`	=	0000 2550	BOX DRAWINGS DOUBLE HORIZONTAL
`ENDCHAR`	(new line)	0000 2029	PARAGRAPH SEPARATOR

6.4.4.5 Other `SPECIAL` characters

Z character	Simple rendering	Code position	Character name
`NLCHAR`	(new line)	0000 000A	LINE FEED
`SPACE`	(space)	0000 0020	SPACE

6.4.5 Symbol characters except mathematical toolkit characters

Math	Code position	Character name
⎸	0000 007C	VERTICAL LINE
&	0000 0026	AMPERSAND
⊢	0000 22A2	RIGHT TACK
∧	0000 2227	LOGICAL AND
∨	0000 2228	LOGICAL OR
⇒	0000 21D2	RIGHTWARDS DOUBLE ARROW
⇔	0000 21D4	LEFT RIGHT DOUBLE ARROW
¬	0000 00AC	NOT SIGN
∀	0000 2200	FOR ALL
∃	0000 2203	THERE EXISTS
×	0000 00D7	MULTIPLICATION SIGN
/	0000 002F	SOLIDUS
=	0000 003D	EQUALS SIGN
∈	0000 2208	ELEMENT OF
:	0000 003A	COLON
;	0000 003B	SEMICOLON
,	0000 002C	COMMA
.	0000 002E	FULL STOP
•	0000 2981	Z NOTATION SPOT
\	0000 29F9	BIG REVERSE SOLIDUS
↾	0000 2A21	Z NOTATION SCHEMA PROJECTION
⨾	0000 2A1F	Z NOTATION SCHEMA COMPOSITION
≫	0000 2A20	Z NOTATION SCHEMA PIPING
+	0000 002B	PLUS SIGN

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

6 Z characters

6.4.6 Mathematical toolkit characters

The mathematical toolkit (annex B) need not be supported by an implementation. If it is supported, it shall use the representations given here.

Mathematical toolkit names that use only Z core language characters, or combinations of Z characters defined here, are not themselves listed here.

6.4.6.1 Section set_toolkit

Math	Code position	Character name
↔	0000 2194	LEFT RIGHT ARROW
→	0000 2192	RIGHTWARDS ARROW
≠	0000 2260	NOT EQUAL TO
∉	0000 2209	NOT AN ELEMENT OF
∅	0000 2205	EMPTY SET
⊆	0000 2286	SUBSET OF OR EQUAL TO
⊂	0000 2282	SUBSET OF
∪	0000 222A	UNION
∩	0000 2229	INTERSECTION
\	0000 005C	REVERSE SOLIDUS
⊖	0000 2296	CIRCLED MINUS
⋃	0000 22C3	N-ARY UNION
⋂	0000 22C2	N-ARY INTERSECTION
𝔽	0001 D53D	MATHEMATICAL DOUBLE-STRUCK CAPITAL F

6.4.6.2 Section relation_toolkit

Math	Code position	Character name
↦	0000 21A6	RIGHTWARDS ARROW FROM BAR
⨾	0000 2A3E	Z NOTATION RELATIONAL COMPOSITION
∘	0000 2218	RING OPERATOR
◁	0000 25C1	WHITE LEFT-POINTING TRIANGLE
▷	0000 25B7	WHITE RIGHT-POINTING TRIANGLE
⩤	0000 2A64	Z NOTATION DOMAIN ANTIRESTRICTION
⩥	0000 2A65	Z NOTATION RANGE ANTIRESTRICTION
∼	0000 223C	TILDE OPERATOR
⦇	0000 2987	Z NOTATION LEFT IMAGE BRACKET
⦈	0000 2988	Z NOTATION RIGHT IMAGE BRACKET
⊕	0000 2295	CIRCLED PLUS

6.4.6.3 Section function_toolkit

Math	Code position	Character name
⇸	0000 21F8	RIGHTWARDS ARROW WITH VERTICAL STROKE
⤔	0000 2914	RIGHTWARDS ARROW WITH TAIL WITH VERTICAL STROKE
↣	0000 21A3	RIGHTWARDS ARROW WITH TAIL
⤀	0000 2900	RIGHTWARDS TWO-HEADED ARROW WITH VERTICAL STROKE
↠	0000 21A0	RIGHTWARDS TWO-HEADED ARROW
⤖	0000 2916	RIGHTWARDS TWO-HEADED ARROW WITH TAIL
⤇	0000 21FB	RIGHTWARDS ARROW WITH DOUBLE VERTICAL STROKE
⤗	0000 2915	RIGHTWARDS ARROW WITH TAIL WITH DOUBLE VERTICAL STROKE

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

6.4.6.4 Section number_toolkit

Math	Code position	Character name
$\mathbb{Z}$	0000 2124	DOUBLE-STRUCK CAPITAL Z
-	0000 002D	HYPHEN-MINUS
$-$	0000 2212	MINUS SIGN
$\leq$	0000 2264	LESS-THAN OR EQUAL TO
$<$	0000 003C	LESS-THAN SIGN
$\geq$	0000 2265	GREATER-THAN OR EQUAL TO
$>$	0000 003E	GREATER-THAN SIGN
$*$	0000 002A	ASTERISK

6.4.6.5 Section sequence_toolkit

Math	Code position	Character name
$\#$	0000 0023	NUMBER SIGN
$\langle$	0000 3008	LEFT ANGLE BRACKET
$\rangle$	0000 3009	RIGHT ANGLE BRACKET
$\frown$	0000 2040	CHARACTER TIE
$\upharpoonleft$	0000 21BF	UPWARDS HARPOON WITH BARB LEFTWARDS
$\upharpoonright$	0000 21BE	UPWARDS HARPOON WITH BARB RIGHTWARDS

6.4.7 Renderings of Z characters

Renderings of Z characters are called glyphs (following the terminology of UCS). The glyphs used for Z characters are device-dependent: a rendering of a Z character on a screen is typically different from its rendering on a piece of paper.

A Z character may also be rendered using different glyphs at different places in a specification, for reasons of emphasis or aesthetics, but such different glyphs all represent the same Z character. For example, '*d*', 'd', '`d`' and '**d**' are all the same Z character. For historical reasons, the following similar-looking glyphs represent different Z characters.

- Schema composition '$\fatsemi$' and the mathematical toolkit character relational composition '$\fatsemi$' are different Z characters.
- Schema projection '$\upharpoonright$' and the mathematical toolkit character filter '$\upharpoonright$' are different Z characters.
- Schema hiding '$\setminus$' and the mathematical toolkit character set minus '$\setminus$' are different Z characters.

7 Lexis

7.1 Introduction

The lexis specifies a function from sequences of Z characters to sequences of tokens. The domain of the function involves all the Z characters of clause 6. The range of the function involves all the tokens used in clause 8. The function is partial: sequences of Z characters that do not conform to the lexis are excluded from consideration at this stage. In each association of a sequence of Z characters with a token, the sequence of Z characters is called the spelling of the token.

The lexis is composed of two parts: a context-free part followed by a context-sensitive part. The former translates the stream of Z characters into a stream of `DECORWORD`s and tokens. The latter classifies each `DECORWORD` as being a keyword, an operator token, or a `NAME` token, taking into account its spelling and the lexical scopes of the operators (see 7.4.4).

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

7.2 Formal definition of context-free lexis

Copyright notice

The reproduction of this clause is permitted on the understanding that this material is public domain, and on the condition that this International Standard is referenced as the source document. With the exception of clauses 6.2, 7.2, 8.2, 11 and annex B, all other parts of the text are subject to the usual copyright rules stated on page ii of this International Standard.

```
TOKENSTREAM      = { SPACE } , { TOKEN , { SPACE } } ;

TOKEN            = DECORWORD | NUMERAL | STROKE
                 | (-tok | )-tok | [-tok | ]-tok | {-tok | }-tok | ⦇ | ⦈ | ⟪ | ⟫
                 | ZED | AX | GENAX | SCH | GENSCH | END | NL
                 ;

DECORWORD        = WORD , { STROKE } ;

WORD             = WORDPART , { WORDPART }
                 | ( LETTER | ( DIGIT — DECIMAL ) ) , ALPHASTR , { WORDPART }
                 | SYMBOL , SYMBOLSTR , { WORDPART }
                 ;

WORDPART         = WORDGLUE , ( ALPHASTR | SYMBOLSTR ) ;
ALPHASTR         = { LETTER | DIGIT } ;
SYMBOLSTR        = { SYMBOL } ;

NUMERAL          = NUMERAL , DECIMAL
                 | DECIMAL
                 ;

STROKE           = STROKECHAR
                 | '↘' , DECIMAL , '↖'
                 ;

(-tok            = '(' ;
)-tok            = ')' ;
[-tok            = '[' ;
]-tok            = ']' ;
{-tok            = '{' ;
}-tok            = '}' ;
⦇                = '⦇' ;
⦈                = '⦈' ;
⟪                = '⟪' ;
⟫                = '⟫' ;

ZED              = ZEDCHAR ;
AX               = AXCHAR ;
SCH              = SCHCHAR ;
GENAX            = AXCHAR , GENCHAR ;
GENSCH           = SCHCHAR , GENCHAR ;
END              = ENDCHAR ;
NL               = NLCHAR ;
```


Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

7.3 Additional lexical restrictions, notes and examples

Words are formed from alphanumeric and symbolic parts.

EXAMPLE 1 The following strings of Z characters are single DECORWORDs: '&+=', 'x_+_y', 'x_+y', 'x^+y', '$x^+{}_+y$'. However, '$x+y$' comprises the three DECORWORDs 'x', '+' and 'y'.

EXAMPLE 2 The following strings of Z characters are single DECORWORDs: 'λS', 'ΔS', '$\exists\times$', '$\exists\_X$', '$\exists_X$'. However, '$\exists X$' is the keyword token '$\exists$', followed by the DECORWORD 'X'.

EXAMPLE 3 The following strings of Z characters are single DECORWORDs: '$\times{:}\in$', '$x\_{:}\_e$', '${}_x{:}_e$'. However, '$x{:}e$' is the word 'x', followed by the keyword token ':', followed by the DECORWORD 'e'.

SPACE is not itself lexed as part of any token. It can be used freely between tokens. The cases where its use is necessary are: between two WORDs, which would otherwise be lexed as a single WORD; between an alphabetic WORD and a NUMERAL, which would otherwise be lexed as a single WORD; between a DECORWORD and a STROKE, which would otherwise be lexed as a single DECORWORD; and between two consecutive NUMERALs, which would otherwise be lexed as a single NUMERAL.

EXAMPLE 4 abc is the DECORWORD 'abc'.
$a\ bc$ is the DECORWORD 'a' followed by the DECORWORD 'bc'.
$a12$ is the DECORWORD '$a12$'.
$a\ 12$ is the DECORWORD 'a' followed by the NUMERAL '12'.
$x!$ is the DECORWORD comprising the WORD 'x' followed by the STROKE '!'.
$x\ !$ is the DECORWORD 'x' followed by the STROKE '!'.
$x!\ !$ is the DECORWORD '$x!$' followed by the STROKE '!'.

The lexis allows a WORD to include subscript digits; it also allows a DECORWORD to be decorated with subscript decimal digits. Trailing subscript decimal digits shall be lexed as strokes, not as part of a WORD.

EXAMPLE 5 x_{a1} is a DECORWORD comprising the WORD 'x_a' and the STROKE '$_1$'.
$x_a?$ is a DECORWORD comprising the WORD 'x_a' and the STROKE '?'.
x_{1a} is a DECORWORD comprising the WORD 'x_{1a}' and no strokes.
a^{b_3} is a DECORWORD comprising the WORD 'a^{b_3}' and no strokes.

A multi-digit last WORDPART enclosed in a $\searrow\ldots\nwarrow$ pair is deprecated, because of the visual ambiguity with multiple STROKE subscript digits.

Throughout subsequent phases of this International Standard, NUMERALs are considered to comprise only the decimal digits enumerated in 6.4.2. Any other decimal digit in a NUMERAL is transformed to the enumerated decimal digit that has the same decimal digit value as defined by UCS.

EXAMPLE MATHEMATICAL DOUBLE-STRUCK DIGIT FIVE, code position 0001 D7DD, has decimal digit value 5 and so, whenever it appears in a NUMERAL, is transformed to DIGIT FIVE, code position 0000 0035.

NOTE 1 Although a parser does not need to know the spelling of particular instances of DECORWORD, NUMERAL, STROKE, etc tokens, subsequent phases of processing do. For example, references to undeclared names are excluded by type inference rules, and the values of numerals may be determined from the values of their decimal digits. The relation between instances of tokens and spellings is not explicitly formalized here. The relation between instances of NUMERALs and their values is formalized for those used as number literal expressions (see 12.2.6.9), because the semantic values of number literal expressions appear in models. The relation between other instances of NUMERALs (those used in tuple selection expressions and those used as operator precedences) and their values is not formalised; these values accord with the usual decimal convention.

Tools may impose restrictions on the forms of names. Section names that are entirely ASCII, alphanumeric, capitalized and short are the most likely to be portable between tools.

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

7.4 Context-sensitive lexis

7.4.1 Introduction

The context-sensitive part of lexis maps each `DECORWORD` to either a keyword token, an operator token, or a `NAME` token. It also strips all `SPACE`s from the token stream, and forwards all other tokens unchanged.

If a `DECORWORD`'s spelling is exactly that of a keyword, the `DECORWORD` is mapped to the corresponding keyword token. Otherwise, if the `DECORWORD`'s `WORD` part's spelling is that of an operator word, the `DECORWORD` is mapped to the relevant operator token. Otherwise, the `DECORWORD` is mapped to a `NAME` token.

In the case of a `NAME` token, for every '↘' `WORDGLUE` character in its `WORD` part, there shall be a paired following '↖' `WORDGLUE` character, for every '↗' `WORDGLUE` character in its `WORD` part, there shall be a paired following '↙' `WORDGLUE` character, and these shall occur only in nested pairs.

NOTE 1 Operators have a similar restriction applied to the whole operator name (see 12.2.8), not to the individual words within the operator.

The keywords are as listed in the following tables. No other spellings give rise to keyword tokens. The columns give:

Spelling: The sequence of Z characters representing the rendering of the token on a high resolution device, such as a bit-mapped screen, or on paper (either hand-written, or printed).

Token: The token used for that keyword in the concrete syntax.

Token name: A suggested form for reading the keyword out loud, suitable for use in reviews, or for discussing specifications over the telephone. In the following, an English language form is given; for other natural languages, other forms may be defined.

NOTE 2 Even where a keyword consists of a single Z character, the token name tends to reflect the keyword's function rather than the form of the Z character.

7.4.2 Alphabetic keywords

Spelling	Token	Token name
else	else	else
false	false	false
function	function	function
generic	generic	generic
if	if	if
leftassoc	leftassoc	left [associative]
let	let	let
ℙ	ℙ	powerset
parents	parents	parents
pre	pre	pre[condition]
relation	relation	relation
rightassoc	rightassoc	right [associative]
section	section	section
then	then	then
true	true	true

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

7.4.3 Symbolic keywords

Spelling	Token	Token name
:	:	colon
==	==	define equal
,	,-tok	comma
::=	::=	free equals
|	|-tok	bar
&	&	and also [free types]
\	\	hide
/	/	rename
.	.	select | dot
;	; -tok	semi[colon]
_	_	arg[ument]
, ,	, ,	list arg[ument]
=	=-tok	equals

EXAMPLE 1 = is recognised as the keyword token =-tok; := is recognised as a `NAME` token; ::= is recognised as the keyword token.

Spelling	Token	Token name
$\vdash ?$	$\vdash ?$	conjecture
$\forall$	$\forall$	for all
$\bullet$	$\bullet$	spot
$\exists$	$\exists$	exists
$\exists_1$	$\exists_1$	unique exists
$\Leftrightarrow$	$\Leftrightarrow$	equivalent | if and only if
$\Rightarrow$	$\Rightarrow$	implies
$\vee$	$\vee$	or
$\wedge$	$\wedge$	and
$\neg$	$\neg$	not
$\in$	$\in$	in | member of | element of
$\upharpoonright$	$\upharpoonright$	project
$\times$	$\times$	cross
λ	λ	lambda
μ	μ	mu
θ	θ	theta
$\mathop{\circ}_{9}$	$\mathop{\circ}_{9}$	schema compose
$\gg$	$\gg$	schema pipe

EXAMPLE 2 $\exists$ and $\exists_1$ ('$\exists$', '$\searrow$', '1', '$\nwarrow$') are recognised as keyword tokens; $\exists_0$ ('$\exists$', '$\searrow$', '0', '$\nwarrow$') is recognised as a `NAME` token.

EXAMPLE 3 λ is recognised as the keyword token; λx is recognised as a `NAME` token; $\lambda\ x$ is recognised as the keyword token followed by a `NAME` token.

7.4.4 User-defined operators

Each operator template creates additional keyword-like associations between `WORD`s and operator tokens. The scope of these associations is the whole of the section in which the operator template appears (not just from the operator template onwards), as well as all sections of which that section is an ancestor, excluding section headers.

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

7 Lexis

NOTE The set of active associations is always a function. This International Standard does not specify how that function is determined: operator template paragraphs provide the information, yet in their concrete syntax it is assumed that the function is already known.

The appropriate token for an operator word is as follows.

PREP	prefix unary relation
PRE	prefix unary function or generic
POSTP	postfix unary relation
POST	postfix unary function or generic
IP	infix binary relation
I	infix binary function or generic
LP	left bracket of non-unary relation
L	left bracket of non-unary function or generic
ELP	first word preceded by expression of non-unary relation
EL	first word preceded by expression of non-unary function or generic
ERP	right bracket preceded by expression of non-unary relation
ER	right bracket preceded by expression of non-unary function or generic
SRP	right bracket preceded by list argument of non-unary relation
SR	right bracket preceded by list argument of non-unary function or generic
EREP	last word followed by expression and preceded by expression of tertiary or higher relation
ERE	last word followed by expression and preceded by expression of tertiary or higher function or generic
SREP	last word followed by expression and preceded by list argument of tertiary or higher relation
SRE	last word followed by expression and preceded by list argument of tertiary or higher function or generic
ES	middle word preceded by expression of non-unary operator
SS	middle word preceded by list argument of non-unary operator

EXAMPLE 1 The operator template paragraph for the (_ + _) operator adds one entry to the mapping.

Spelling	**Token**
+	I

EXAMPLE 2 The operator template paragraph for the (_ ⦇ _ ⦈) operator adds two entries to the mapping.

Spelling	**Token**
⦇	EL
⦈	ER

EXAMPLE 3 The operator template paragraph for the (*disjoint* _) operator adds one entry to the mapping.

Spelling	**Token**
disjoint	PREP

EXAMPLE 4 The operator template paragraph for the (⟨ _ ⟩) operator adds two entries to the mapping.

Spelling	**Token**
⟨	L
⟩	SR

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

7.5 Newlines

The Z character `NLCHAR` is lexed either as a token separator (like the `SPACE` character) or as the token `NL`, depending on its context. A *soft newline* is a `NLCHAR` that is lexed as a token separator. A *hard newline* is a `NLCHAR` that is lexed as a `NL` token.

Tokens are assigned to a *newline category*, namely BOTH, AFTER, BEFORE or NEITHER, based on whether that token could start or end a Z phrase.

- BOTH: newlines are soft before and after the token, because it is infix, something else has to appear before it and after it.

  else function generic leftassoc parents relation rightassoc section then
  ::= |-tok ⦇ ⦈ & ⊢? , , ∧ ∨ ⇒ ⇔ × / =-tok ∈ == : ; -tok ,-tok . • \ ↾ ⨾ ≫
  `I` `IP` `EL` `ELP` `ERE` `EREP` `ES` `SS` `SRE` `SREP`

  All newlines are soft outside of a `DeclPart` or a `Predicate`.

  NOTE Tokens that cannot appear in these contexts are in category BOTH. This includes the box tokens. Newlines at the very beginning or end of a specification are soft.

- AFTER: newlines are soft after the token, because it is prefix, something else has to appear after it.

  if let pre
  [−tok _ ¬ ∀ ∃ $\exists_1$ ℙ (−tok {−tok ⦇ λ μ θ
  `PRE` `PREP` `L` `LP`

- BEFORE: newlines are soft before the token, because it is postfix, something else has to appear before it.

  ]−tok)−tok }−tok ⦈
  `POST` `POSTP` `ER` `ERP` `SR` `SRP`

- NEITHER: no newlines are soft, because such a token is nofix, nothing else need appear before or after it.

  false true
  `NAME` `NUMERAL` `STROKE`

For each `NLCHAR`, the newline categories of the closest token generated from the preceding Z characters and the token generated from the immediately following Z characters are examined. If either token allows the newline to be soft in that position, it is soft, otherwise it is hard (and hence recognised as a `NL` token).

The operator template paragraph allows the definition of various mixfix names (see 7.4.4), which are placed in the appropriate newline category. Other (ordinary) user declared names are nofix, and so are placed in NEITHER.

Consecutive `NLCHAR`s are treated the same as a single `NLCHAR`.

8 Concrete syntax

8.1 Introduction

The concrete syntax defines the syntax of the Z language: every sentence of the Z language is recognised by this syntax, and all sentences recognised by this syntax are sentences of the Z language. The concrete syntax is written in terms of the tokens generated by the lexis (clause 7). There are no terminal symbols within this syntax, so as to establish a formal connection with that lexis. Sequences of tokens that are not recognised by this syntax are not sentences of the Z language and are thus excluded from consideration by subsequent phases and so are not given a semantics by this International Standard.

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

8 Concrete syntax

A parser conforming to this concrete syntax converts a concrete sentence to a parse tree.

The non-terminal symbols of the concrete syntax that are written as mathematical symbols or are entirely `CAPITALIZED` or Roman are Z tokens defined in the lexis (claue 7). The other non-terminal symbols are written in `MixedCase` and are defined within the concrete syntax.

8.2 Formal definition of concrete syntax

Copyright notice

The reproduction of this clause is permitted on the understanding that this material is public domain, and on the condition that this International Standard is referenced as the source document. With the exception of clauses 6.2, 7.2, 8.2, 11 and annex B, all other parts of the text are subject to the usual copyright rules stated on page ii of this International Standard.

```
Specification    = { Section }                                        (* sectioned specification *)
                 | { Paragraph }                                      (* anonymous specification *)
                 ;

Section          = ZED , section , NAME , parents , [ NAME , { ,-tok , NAME } ] , END ,
                       { Paragraph }                                  (* inheriting section *)
                 | ZED , section , NAME , END , { Paragraph }         (* base section *)
                 ;

Paragraph        = ZED , [-tok , NAME , { ,-tok , NAME } , ]-tok , END   (* given types *)
                 | AX , SchemaText , END                              (* axiomatic description *)
                 | SCH , NAME , SchemaText , END                      (* schema definition *)
                 | GENAX , [-tok , Formals , ]-tok , SchemaText , END
                                                                      (* generic axiomatic description *)
                 | GENSCH , NAME , [-tok , Formals , ]-tok , SchemaText , END
                                                                      (* generic schema definition *)
                 | ZED , DeclName , == , Expression , END             (* horizontal definition *)
                 | ZED , NAME , [-tok , Formals , ]-tok , == , Expression , END
                                                                      (* generic horizontal definition *)
                 | ZED , GenName , == , Expression , END              (* generic operator definition *)
                 | ZED , Freetype , { & , Freetype } , END            (* free types *)
                 | ZED , ⊢? , Predicate , END                         (* conjecture *)
                 | ZED , [-tok , Formals , ]-tok , ⊢? , Predicate , END   (* generic conjecture *)
                 | ZED , OperatorTemplate , END                       (* operator template *)
                 ;

Freetype         = NAME , ::= , Branch , { |-tok , Branch } ;         (* free type *)

Branch           = DeclName , [ ⟪ , Expression , ⟫ ] ;                (* element or injection *)

Formals          = NAME , { ,-tok , NAME } ;                          (* generic parameters *)

Predicate        = Predicate , NL , Predicate                         (* newline conjunction *)
                 | Predicate , ;-tok , Predicate                      (* semicolon conjunction *)
                 | ∀ , SchemaText , • , Predicate                     (* universal quantification *)
                 | ∃ , SchemaText , • , Predicate                     (* existential quantification *)
                 | ∃₁ , SchemaText , • , Predicate                    (* unique existential quantification *)
                 | Predicate , ⇔ , Predicate                          (* equivalence *)
                 | Predicate , ⇒ , Predicate                          (* implication *)
                 | Predicate , ∨ , Predicate                          (* disjunction *)
                 | Predicate , ∧ , Predicate                          (* conjunction *)
                 | ¬ , Predicate                                      (* negation *)
```


Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

```
                | Relation                                        (* relation operator application *)
                | Expression                                      (* schema predicate *)
                | true                                            (* truth *)
                | false                                           (* falsity *)
                | (-tok , Predicate , )-tok                       (* parenthesized predicate *)
                ;

Expression      = ∀ , SchemaText , • , Expression                 (* schema universal quantification *)
                | ∃ , SchemaText , • , Expression                 (* schema existential quantification *)
                | ∃₁ , SchemaText , • , Expression                (* schema unique existential quantification *)
                | λ , SchemaText , • , Expression                 (* function construction *)
                | μ , SchemaText , • , Expression                 (* definite description *)
                | let , DeclName , == , Expression ,
                      { ;-tok , DeclName , == , Expression } ,
                          • , Expression                          (* substitution expression *)
                | Expression , ⇔ , Expression                     (* schema equivalence *)
                | Expression , ⇒ , Expression                     (* schema implication *)
                | Expression , ∨ , Expression                     (* schema disjunction *)
                | Expression , ∧ , Expression                     (* schema conjunction *)
                | ¬ , Expression                                  (* schema negation *)
                | if , Predicate , then , Expression , else , Expression   (* conditional *)
                | Expression , ⨾ , Expression                     (* schema composition *)
                | Expression , ≫ , Expression                     (* schema piping *)
                | Expression , \ , (-tok , DeclName , { ,-tok , DeclName } , )-tok
                                                                  (* schema hiding *)
                | Expression , ↾ , Expression                     (* schema projection *)
                | pre , Expression                                (* schema precondition *)
                | Expression , × , Expression , { × , Expression }   (* Cartesian product *)
                | ℙ , Expression                                  (* powerset *)
                | Application                                     (* function and generic operator application *)
                | Expression , Expression                         (* application *)
                | Expression , STROKE                             (* schema decoration *)
                | Expression , [-tok , DeclName , / , DeclName ,
                      { ,-tok , DeclName , / , DeclName } , ]-tok (* schema renaming *)
                | Expression , . , RefName                        (* binding selection *)
                | Expression , . , NUMERAL                        (* tuple selection *)
                | θ , Expression , { STROKE }                     (* binding construction *)
                | RefName                                         (* reference *)
                | RefName , [-tok , Expression , { ,-tok , Expression } , ]-tok
                                                                  (* generic instantiation *)
                | NUMERAL                                         (* number literal *)
                | {-tok , [ Expression , { ,-tok , Expression } ] , }-tok   (* set extension *)
                | {-tok , SchemaText , • , Expression , }-tok     (* set comprehension *)
                | ( ( {-tok , SchemaText , }-tok )   — ( {-tok , }-tok ) )
                                  — ( {-tok , Expression , }-tok )
                                                                  (* characteristic set comprehension *)
                | ( [-tok , SchemaText , ]-tok )   — ( [-tok , Expression , ]-tok )
                                                                  (* schema construction *)
                | ⦇ , [ DeclName , == , Expression ,
                      { ,-tok , DeclName , == , Expression } ] , ⦈   (* binding extension *)
                | (-tok , Expression , ,-tok , Expression , { ,-tok , Expression } , )-tok
                                                                  (* tuple extension *)
                | (-tok , μ , SchemaText , )-tok                  (* characteristic definite description *)
```


Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

8 Concrete syntax

```
                  | (-tok , Expression , )-tok                    (* parenthesized expression *)
                  ;

SchemaText        = [ DeclPart ] , [ |-tok , Predicate ] ;

DeclPart          = Declaration , { ( ;-tok | NL ) , Declaration } ;

Declaration       = DeclName , { ,-tok , DeclName } , : , Expression
                  | DeclName , == , Expression
                  | Expression
                  ;

OperatorTemplate  = relation , Template
                  | function , CategoryTemplate
                  | generic , CategoryTemplate
                  ;

CategoryTemplate  = PrefixTemplate
                  | PostfixTemplate
                  | Prec , Assoc , InfixTemplate
                  | NofixTemplate
                  ;

Prec              = NUMERAL ;

Assoc             = leftassoc
                  | rightassoc
                  ;

Template          = PrefixTemplate
                  | PostfixTemplate
                  | InfixTemplate
                  | NofixTemplate
                  ;

PrefixTemplate    = (-tok , PrefixName , )-tok
                  | (-tok , ℙ , _ , )-tok
                  ;

PostfixTemplate   = (-tok , PostfixName , )-tok ;

InfixTemplate     = (-tok , InfixName , )-tok ;

NofixTemplate     = (-tok , NofixName , )-tok ;

DeclName          = NAME
                  | OpName
                  ;

RefName           = NAME
                  | (-tok , OpName , )-tok
                  ;

OpName            = PrefixName
                  | PostfixName
                  | InfixName
                  | NofixName
                  ;
```


Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

```
PrefixName        = PRE , _
                  | PREP , _
                  | L , { _ , ES | ,, , SS } , ( _ , ERE | ,, , SRE ) , _
                  | LP , { _ , ES | ,, , SS } , ( _ , EREP | ,, , SREP ) , _
                  ;

PostfixName       = _ , POST
                  | _ , POSTP
                  | _ , EL , { _ , ES | ,, , SS } , ( _ , ER | ,, , SR )
                  | _ , ELP , { _ , ES | ,, , SS } , ( _ , ERP | ,, , SRP )
                  ;

InfixName         = _ , I , _
                  | _ , IP , _
                  | _ , EL , { _ , ES | ,, , SS } , ( _ , ERE | ,, , SRE ) , _
                  | _ , ELP , { _ , ES | ,, , SS } , ( _ , EREP | ,, , SREP ) , _
                  ;

NofixName         = L , { _ , ES | ,, , SS } , ( _ , ER | ,, , SR )
                  | LP , { _ , ES | ,, , SS } , ( _ , ERP | ,, , SRP )
                  ;

GenName           = PrefixGenName
                  | PostfixGenName
                  | InfixGenName
                  | NofixGenName
                  ;

PrefixGenName     = PRE , NAME
                  | L , { NAME , ( ES | SS ) } , NAME , ( ERE | SRE ) , NAME
                  ;

PostfixGenName    = NAME , POST
                  | NAME , EL , { NAME , ( ES | SS ) } , NAME , ( ER | SR )
                  ;

InfixGenName      = NAME , I , NAME
                  | NAME , EL , { NAME , ( ES | SS ) } , NAME , ( ERE | SRE ) , NAME
                  ;

NofixGenName      = L , { NAME , ( ES | SS ) } , NAME , ( ER | SR ) ;

Relation          = PrefixRel
                  | PostfixRel
                  | InfixRel
                  | NofixRel
                  ;

PrefixRel         = PREP , Expression
                  | LP , ExpSep , ( Expression , EREP | ExpressionList , SREP ) , Expression
                  ;

PostfixRel        = Expression , POSTP
                  | Expression , ELP , ExpSep , ( Expression , ERP | ExpressionList , SRP )
                  ;
```


Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

8 Concrete syntax

```
InfixRel          = Expression , ( ∈ | =-tok | IP ) , Expression ,
                        { ( ∈ | =-tok | IP ) , Expression }
                  | Expression , ELP , ExpSep ,
                      ( Expression , EREP | ExpressionList , SREP ) , Expression
                  ;

NofixRel          = LP , ExpSep , ( Expression , ERP | ExpressionList , SRP ) ;

Application       = PrefixApp
                  | PostfixApp
                  | InfixApp
                  | NofixApp
                  ;

PrefixApp         = PRE , Expression
                  | L , ExpSep , ( Expression , ERE | ExpressionList , SRE ) , Expression
                  ;

PostfixApp        = Expression , POST
                  | Expression , EL , ExpSep , ( Expression , ER | ExpressionList , SR )
                  ;

InfixApp          = Expression , I , Expression
                  | Expression , EL , ExpSep ,
                      ( Expression , ERE | ExpressionList , SRE ) , Expression
                  ;

NofixApp          = L , ExpSep , ( Expression , ER | ExpressionList , SR ) ;

ExpSep            = { Expression , ES | ExpressionList , SS } ;

ExpressionList    = [ Expression , { ,-tok , Expression } ] ;
```

8.3 User-defined operators

A template can reuse the same words as other templates in the same scope, so long as the same token is associated with a word by all templates that use that word. A template's whole operator name shall be different from those of other templates in the same scope. All templates in a scope that use the same word shall have the same precedence.

EXAMPLE 1 These two templates reuse *a* acceptably.

generic 30 leftassoc (_ *a* _ *b* _)
generic 30 leftassoc (_ *a* _ *c* _)

EXAMPLE 2 These two templates conflict unacceptably, because the whole operator names are identical.

function 10 leftassoc (_ *d* _)
generic 20 leftassoc (_ *d* _)

EXAMPLE 3 These two templates conflict unacceptably, because they reuse *a* and have different precedences.

generic 30 leftassoc (_ *a* _ *b* _)
generic 40 leftassoc (_ *a* _ *c* _)

In the `Template` rule's auxiliaries, the name of each of the operator tokens shall not have any `STROKE`s.

NOTE 1 This is so that any common decoration of those words can be treated as an application of a decorated instance of that operator.

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

All prefix operators are right associative. All postfix operators are left associative. Different associativities shall not be used at the same precedence level by operator template paragraphs in the same scope.

EXAMPLE 4 These two templates conflict unacceptably, because they have the same precedence but different associativities.

function 50 leftassoc $(\_\ e\ \_)$
function 50 rightassoc $(\_\ f\ \_)$

Table 31 defines the relative precedences of the productions of `Expression` and `Predicate`. The rows in the table are ordered so that the entries with higher precedence (and so that bind more strongly) appear nearer the top of the table than those with lower precedence (that bind more weakly). Associativity has significance only in determining the nesting of applications involving non-associative operators of the same precedence. Explicitly-defined infix function and generic operator applications have a range of precedences specified numerically in the corresponding operator template paragraph. Cartesian product expressions have precedence value 8 within that numeric range.

NOTE 2 The order of productions in the `Predicate` and `Expression` rules is based roughly on the precedences of their operators. Some productions have the same precedence as their neighbours, and so the separate table of operator precedences is necessary.

NOTE 3 One way of parsing nested operator applications at different user-defined levels of precedence and associativity is explained by Lalonde and des Rivieres [9]. Lalonde and des Rivieres approach is to "use a grammar ... that describes expressions without regard to the precedence [and associativity] of the operators", and then to transform the resulting parse tree to take account of precedence and associativity. Assuming the grammar uses left associativity for all operators, the transformations to take into account precedence and associativity where necessary are as follows.

$$\begin{aligned}(e_1\ \mathit{infix}_1\ e_2)\ \mathit{infix}_2\ e_3 &\Longrightarrow e_1\ \mathit{infix}_1\ (e_2\ \mathit{infix}_2\ e_3)\\ (e_1\ \mathit{infix}\ e_2)\ \mathit{post} &\Longrightarrow e_1\ \mathit{infix}\ (e_2\ \mathit{post})\\ (\mathit{pre}\ e)\ \mathit{post} &\Longrightarrow \mathit{pre}\ (e\ \mathit{post})\end{aligned}$$

If right associativity was instead assumed, then the following transformations would be needed.

$$\begin{aligned}e_1\ \mathit{infix}_1\ (e_2\ \mathit{infix}_2\ e_3) &\Longrightarrow (e_1\ \mathit{infix}_1\ e_2)\ \mathit{infix}_2\ e_3\\ \mathit{pre}\ (e_1\ \mathit{infix}\ e_2) &\Longrightarrow (\mathit{pre}\ e_1)\ \mathit{infix}\ e_2\end{aligned}$$

Using distinct variants of the operator tokens `PRE`|...|`SS` for relational operators from those for function and generic operators allows these transformations to avoid dealing with those notations whose precedences lie between the relations and the functions, such as the schema operations.

NOTE 4 In the `PrefixTemplate` rule, the production for powerset enables explicit definition of $\mathbb{P}_1$ in the mathematical toolkit to coexist with the treatment of $\mathbb{P}$ as a keyword by the lexis.

8.4 Additional syntactic restrictions and notes

`STROKE` is used in three contexts: within `NAME`s, in binding construction expressions, and in schema decoration expressions. The condition for a `STROKE` to be considered as part of a `NAME` was given in 7.3. Other `STROKE`s are considered to be parts of binding construction expressions if they can be when interpreted from left to right. The schema decoration expression interpretation is considered last.

EXAMPLE 1 In $\theta S'\ '\ '$, the first $'$ is part of the `NAME` S', the second $'$ can be part of the binding construction expression, and the third $'$ is syntactically part of a schema decoration expression, though that will be rejected by the type inference rules as only schemas can be decorated, not bindings.

A predicate can be just an expression, yet the same logical operators ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$, $\forall$, $\exists$, $\exists_1$) can be used in both expressions and predicates. Where a predicate is expected, and one of these logical operators is used on expressions, there is an ambiguity: either the whole logical operation is an expression and that expression is used

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

8 Concrete syntax

Table 31 — Operator precedences and associativities

Productions	Associativity
binding construction	
binding selection, tuple selection	
schema renaming	
schema decoration	
application	left
postfix function and generic operator application	
powerset, prefix function and generic operator application	
Cartesian product, infix function and generic operator application	
schema precondition	
schema projection	left
schema hiding	
schema piping	
schema composition	
conditional	
substitution expression	
definite description	
function construction	
relation operator application	
negation	
conjunction	
disjunction	
implication	right
equivalence	
universal, existential and unique existential quantifications	
newline conjunction, semicolon conjunction	

as a predicate, or the whole logical operation is a predicate involving expressions each used as a predicate. This ambiguity is benign, as both interpretations have the same precedence, associativity and meaning.

A section header shall be parsed in its entirety before bringing the declarations and operator templates of its parent sections into scope.

NOTE 1 This prevents surprises when the name of a parent section is the same as the name of an operator defined in another parent.

Application expressions' lack of any keyword token between their juxtaposed expressions is a problem for some implementations of parsers, which may require the parentheses in phrases such as $f\ (seq\ X)$.

NOTE 2 The juxtaposition of two expressions $e_1 e_2$ is always parsed as the application of function e_1 to argument e_2, never as the application of relation e_1 to argument e_2 which in some previous dialects of Z, e.g. King *et al* [8], was equivalent to the relation $e_2 \in e_1$. In Standard Z, membership is the normal form (canonical representation) of all relational predicates (see 12.2.10), and juxtaposition is the normal form of all application expressions (see 12.2.11).

NOTE 3 The syntax of conjectures is deliberately simple. This is so as to be compatible with the syntaxes of sequents as found in as many different theorem provers as possible, while establishing a common form to enable interchange.

NOTE 4 Implementations of parsers commonly inspect the next token from the input stream and have to decide there and then whether that token is another token in an incomplete phrase or whether the current phrase is complete and the token is starting a new phrase. The tokens defined by the lexis (clause 7) are insufficient for such an implementation of a parser.

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

EXAMPLE 2 At the first comma in the set extension $\{x, y, z\}$ the x shall be seen to be an expression, yet the phrase might yet turn out to be the set comprehension $\{x, y, z : e\}$.

EXAMPLE 3 At the opening square bracket in the application to a schema construction $i\ [e_1;\ e_2]$ the name i shall be seen to be an expression, yet the phrase might yet turn out to be the generic instantiation $i[e_1]$.

One solution to such problems is to try all possible parses and accept the one that works. Another solution is to have the lexer lookahead far enough to be able to distinguish which of the alternative cases is present, and to provide the parser with one of several distinct tokens. Expressions $\{x, y, z\}$ and $\{x, y, z : e\}$ can be distinguished by looking ahead from a comma, over following alternating names (including operator names) and commas, for a : or == token, and using distinct comma tokens depending on whether that is seen or not. Expressions $i\ [e_1;\ e_2]$ and $i[e_1]$ can be distinguished by looking ahead from the open square bracket for the matching closing square bracket, stopping if a ; -tok, :, == or |-tok token is encountered, and using distinct open square bracket tokens for the matched and stopped cases.

NOTE 5 An `ExpressionList` phrase will be regarded as an expression whose value is a sequence (see 12.2.12). In defining operators that take such `ExpressionList`s as arguments, it is convenient to have the operations of *sequence_toolkit* (see B.8) in scope.

NOTE 6 The `ExpressionList` rule is used only in operator applications, not in set extension, tuple extension and generic instantiation expressions, so that the syntactic transformation in 12.2.12 is applied only to operator applications.

8.5 Box renderings

There are two different sets of box renderings in widespread use, as illustrated here. Any particular presentation of a section shall use one set or the other throughout. The middle line shall be omitted when the paragraph has no predicates, but otherwise shall be retained if the paragraph has no declarations. The outlines need be only as wide as the text, but are here shown as wide as the page.

8.5.1 First box rendering

The following four paragraphs illustrate the first of two alternative renderings of box tokens.

An axiomatic paragraph, involving the `AX`, |-tok and `END` tokens, shall have this box rendering.

```
| DeclPart
|-----------
| Predicate
```

A schema paragraph, involving the `SCH`, |-tok and `END` tokens, shall have this box rendering.

```
NAME
| DeclPart
|-----------
| Predicate
```

A generic axiomatic paragraph, involving the `GENAX`, |-tok and `END` tokens, shall have this box rendering.

```
[Formals]
| DeclPart
|-----------
| Predicate
```

A generic schema paragraph, involving the `GENSCH`, |-tok and `END` tokens, shall have this box rendering.

```
NAME [Formals]
| DeclPart
|-----------
| Predicate
```


Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

8.5.2 Second box rendering

The following four paragraphs illustrate the second of two alternative renderings of box tokens.

An axiomatic paragraph, involving the `AX`, |-tok and `END` tokens, shall have this box rendering.

```
| DeclPart
|-----------
| Predicate
```

A schema paragraph, involving the `SCH`, |-tok and `END` tokens, shall have this box rendering.

```
NAME
| DeclPart
|-----------
| Predicate
```

A generic axiomatic paragraph, involving the `GENAX`, |-tok and `END` tokens, shall have this box rendering.

```
[Formals]
| DeclPart
|-----------
| Predicate
```

A generic schema paragraph, involving the `GENSCH`, |-tok and `END` tokens, shall have this box rendering.

```
NAME [Formals]
| DeclPart
|-----------
| Predicate
```

9 Characterisation rules

9.1 Introduction

The characterisation rules together map the parse tree of a concrete syntax sentence to the parse tree of an equivalent concrete syntax sentence in which all implicit characteristic tuples have been made explicit.

Only concrete trees that are mapped to different trees are given explicit characterisation rules. The characterisation rules are listed in the same order as the corresponding productions of the concrete syntax.

Characteristic tuples are calculated from schema texts by the metalanguage function *chartuple* (see 9.2).

9.2 Characteristic tuple

A characteristic tuple is computed in two phases: *charac*, which returns a sequence of expressions, and *mktuple*, which converts that sequence into the characteristic tuple.

$$chartuple\ t \quad = \quad mktuple\ (charac\ t)$$

Sequences of expressions are enclosed between metalanguage brackets $\langle$ and $\rangle$, in general $\langle e_1, ..., e_n \rangle$. Two sequences of expressions are concatenated by the $\frown$ operator.

$$\langle e_1, ..., e_n \rangle \frown \langle e_{n+1}, ..., e_{n+m} \rangle \quad = \quad \langle e_1, ..., e_n, e_{n+1}, ..., e_{n+m} \rangle$$

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 13568:2002

$$
\begin{aligned}
charac\ (d_1;\ ...;\ d_n \mid p) &= charac\ (d_1;\ ...;\ d_n) \\
charac\ (d_1;\ ...;\ d_n) &= charac\ d_1 \frown ... \frown charac\ d_n \qquad where\ n \geq 1 \\
charac\ () &= \langle\!\langle\ \rangle\!\rangle \\
charac\ (i_1, ..., i_n : e) &= \langle i_1, ..., i_n \rangle \\
charac\ (i == e) &= \langle i \rangle \\
charac\ (e\ ^*) &= \langle \theta\, e^* \rangle
\end{aligned}
$$

$$
\begin{aligned}
mktuple\ \langle e \rangle &= e \\
mktuple\ \langle e_1, ..., e_n \rangle &= (e_1, ..., e_n) \qquad where\ n \geq 2
\end{aligned}
$$

NOTE 1 In the last case of *charac*, the type inference rule in 13.2.6.9 ensures that e is a schema.

NOTE 2 In *mktuple*, the result is a Z expression, so the brackets in its second equation are those of a tuple extension.

NOTE 3 The characteristic tuple operation determines a new phrase for use in a larger phrase. It does not manipulate semantic values or type values. It is more akin to a transformation rule in that it operates on phrases, but unlike a transformation rule it does not replace a phrase by an equivalent phrase. Its definition needs auxiliaries for representing sequences of phrases and concatenating sequences of phrases. The sequence brackets defined in 4.2.11 are not suitable, as they manipulate metalanguage sequences. Rather than invent yet more notation, the usual Z-like notations for sequences are redefined just for this definition.

9.3 Formal definition of characterisation rules

9.3.1 Function construction expression

The value of the function construction expression $\lambda\ t \bullet e$ is the function associating values of the characteristic tuple of t with corresponding values of e.

$$\lambda\ t \bullet e \quad \Longrightarrow \quad \{t \bullet (chartuple\ t, e)\}$$

It is semantically equivalent to the set of pairs representation of that function.

9.3.2 Characteristic set comprehension expression

The value of the characteristic set comprehension expression $\{t\}$ is the set of the values of the characteristic tuple of t.

$$\{t\} \quad \Longrightarrow \quad \{t \bullet chartuple\ t\}$$

It is semantically equivalent to the corresponding set comprehension expression in which the characteristic tuple is made explicit.

9.3.3 Characteristic definite description expression

The value of the characteristic definite description expression $(\mu\ t)$ is the sole value of the characteristic tuple of schema text t.

$$(\mu\ t) \quad \Longrightarrow \quad \mu\ t \bullet chartuple\ t$$

It is semantically equivalent to the corresponding definite description expression in which the characteristic tuple is made explicit.

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

10 Annotated syntax

10.1 Introduction

The annotated syntax defines a language that includes all sentences that could be produced by application of the syntactic transformation rules (clause 12) to sentences of the concrete syntax (clause 8). This language's set of sentences would be a subset of that defined by the concrete syntax but for introduction of type annotations and use of expressions in place of schema texts.

Like the concrete syntax, this annotated syntax is written in terms of the tokens generated by the lexis; there are no terminal symbols within this syntax. The added type annotation notation uses three tokens not defined in the lexis: `GIVEN`, `GENTYPE`, and ⦂.

Some additional characters that are presumed to be distinct from the characters used in concrete phrases are introduced as follows. The constituent words of operators are glued together to form single names using the character ⋈, which belongs to the `WORDGLUE` character class. (These single names cannot clash with any existing names.) For comparing the bindings of two schemas, one schema is decorated using the character ⋊, which belongs to the `STROKECHAR` character class. (Inclusion of a schema so decorated cannot capture any existing references.) For defining the semantics of types, the names of given sets and generic parameters are decorated with characters ♡ and ♠ respectively, both of which belong to the `STROKECHAR` character class. (This avoids local declarations of the same names making holes in the scopes of the types.)

There are no parentheses in the annotated syntax as defined here. A sentence or phrase of the annotated syntax should be thought of as a tree structure of nested formulae. When presented as linear text, however, the precedences of the concrete syntax may be assumed and parentheses may be inserted to override those precedences. The precedence of the type annotation ⦂ operator is then weaker than all other operators, and the precedences and associativities of the type notations are analogous to those of the concrete notations of similar appearance.

NOTE 1 This annotated syntax permits some verification of the syntactic transformation rules to be performed.

NOTE 2 The annotated syntax is similar to an annotated tree (abstract syntax) used in a tool, but the level of abstraction effected by the characterization rules and syntactic transformation rules might not be appropriate for a tool.

10.2 Formal definition of annotated syntax

```
Specification     = { Section } ;                                   (* sectioned specification *)

Section           = ZED , section , NAME , parents , [ NAME , { ,-tok , NAME } ] , END ,
                        { Paragraph } , [ ⦂ , SectTypeEnv ] ;        (* inheriting section *)

Paragraph         = ZED , [-tok , NAME , { ,-tok , NAME } , ]-tok , END ,
                        [ ⦂ , Signature ]                                  (* given types *)
                  | AX , Expression , END ,
                        [ ⦂ , Signature ]                          (* axiomatic description *)
                  | GENAX , [-tok , NAME , { ,-tok , NAME } , ]-tok , Expression , END ,
                        [ ⦂ , Signature ]                   (* generic axiomatic description *)
                  | ZED , NAME , ::= , NAME , [ ⟪ , Expression , ⟫ ] ,
                      { |-tok , NAME , [ ⟪ , Expression , ⟫ ] } ,
                      { & , NAME , ::= , NAME , [ ⟪ , Expression , ⟫ ] ,
                      { |-tok , NAME , [ ⟪ , Expression , ⟫ ] } } , END ,
                        [ ⦂ , Signature ]                                 (* free types *)
                  | ZED , ⊢? , Predicate , END ,
                        [ ⦂ , Signature ]                                (* conjecture *)
                  | ZED , [-tok , NAME , { ,-tok , NAME } , ]-tok , ⊢? , Predicate , END ,
                        [ ⦂ , Signature ]                          (* generic conjecture *)
                  ;
```


Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002

```
Predicate        = Expression , ∈ , Expression                          (* membership *)
                 | true                                                 (* truth *)
                 | ¬ , Predicate                                        (* negation *)
                 | Predicate , ∧ , Predicate                            (* conjunction *)
                 | ∀ , Expression , • , Predicate                       (* universal quantification *)
                 | ∃1 , Expression , • , Predicate                      (* unique existential quantification *)
                 ;

Expression       = NAME ,
                       [ ⦂ , Type ]                                     (* reference *)
                 | NAME , [-tok , Expression , { ,-tok , Expression } , ]-tok ,
                       [ ⦂ , Type ]                                     (* generic instantiation *)
                 | {-tok , [ Expression , { ,-tok , Expression } ] , }-tok ,
                       [ ⦂ , Type ]                                     (* set extension *)
                 | {-tok , Expression , • , Expression , }-tok ,
                       [ ⦂ , Type ]                                     (* set comprehension *)
                 | ℙ , Expression ,
                       [ ⦂ , Type ]                                     (* powerset *)
                 | (-tok , Expression , ,-tok , Expression , { ,-tok , Expression } , )-tok ,
                       [ ⦂ , Type ]                                     (* tuple extension *)
                 | Expression , . , NUMERAL ,
                       [ ⦂ , Type ]                                     (* tuple selection *)
                 | ⦇ , NAME , == , Expression ,
                       { ,-tok , NAME , == , Expression } , ⦈ ,
                           [ ⦂ , Type ]                                 (* binding extension *)
                 | θ , Expression , { STROKE } ,
                       [ ⦂ , Type ]                                     (* binding construction *)
                 | Expression , . , NAME ,
                       [ ⦂ , Type ]                                     (* binding selection *)
                 | Expression , Expression ,
                       [ ⦂ , Type ]                                     (* application *)
                 | μ , Expression , • , Expression ,
                       [ ⦂ , Type ]                                     (* definite description *)
                 | [-tok , NAME , : , Expression , ]-tok ,
                       [ ⦂ , Type ]                                     (* variable construction *)
                 | [-tok , Expression , |-tok , Predicate , ]-tok ,
                       [ ⦂ , Type ]                                     (* schema construction *)
                 | ¬ , Expression ,
                       [ ⦂ , Type ]                                     (* schema negation *)
                 | Expression , ∧ , Expression ,
                       [ ⦂ , Type ]                                     (* schema conjunction *)
                 | Expression , \ , (-tok , NAME , { ,-tok , NAME } , )-tok ,
                       [ ⦂ , Type ]                                     (* schema hiding *)
                 | ∀ , Expression , • , Expression ,
                       [ ⦂ , Type ]                                     (* schema universal quantification *)
                 | ∃1 , Expression , • , Expression ,
                       [ ⦂ , Type ]                                     (* schema unique existential quantification *)
                 | Expression , [-tok , NAME , / , NAME ,
                       { ,-tok , NAME , / , NAME } , ]-tok ,
                           [ ⦂ , Type ]                                 (* schema renaming *)
                 | pre , Expression ,
                       [ ⦂ , Type ]                                     (* schema precondition *)
```


Documents
Document Sharing Hub
Full document details: BS ISO/IEC 13568:2002