Characters + Mark-up = Z Lexis

Ian Toyn[1] and Susan Stepney[2]

[1] Department of Computer Science, University of York,
Heslington, York, YO10 5DD, UK.
ian@cs.york.ac.uk
[2] Logica UK Ltd,
Betjeman House, 104 Hills Road, Cambridge, CB2 1LQ, UK.
stepneys@logica.com

Abstract. The mathematical symbols in Z have caused problems for users and tool builders in the past—precisely what is allowed? ISO Standard Z answers this question. This paper considers the Z notation at the level of the individual characters that make up a specification. For Z authors: it reviews the internationalisation of Z, discusses what characters can be used in forming names, and summarises the changes made to LaTeX mark-up in ISO Standard Z. For Z tool builders: it explains the sequence of processing that is prerequisite to the lexing of a Standard Z specification, and considers in detail the processing of LaTeX mark-up.

1 Introduction

Consider a typical paragraph from a Z specification.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{\Phi Update}} \\
\hline
\Delta System \\
\Delta File \\
f? : ID \\
\hline
f? \mapsto \theta\ File \in fs \\
fs' = fs \oplus \{f? \mapsto \theta File~'\} \\
\hline
\end{array}
$$

It exhibits several characteristic features: the outline around the mathematics, the use of Greek letters (Δ and θ) as part of the core language, mathematical symbols ($\oplus$ and $\mapsto$) from the standard mathematical toolkit, and the use of a further Greek letter (Φ) introduced by the specifier.

Such sophisticated orthography can be easily written using a pen, pencil or chalk, but poses problems for ASCII-based keyboards. Software tools for Z have resorted to using *mark-up languages*, in which sequences of ASCII characters are used to encode phrases of a Z specification. Mark-up languages have been devised for representing Z specifications embedded in LaTeX [Spivey 1992a, King 1990], troff [Toyn], e-mail [ISO-Z], SGML [Germán *et al.* 1994] and XML [Ciancarini *et al.* 1998] documents, amongst many others. For example, the schema above could be marked-up in LaTeX as

```
\begin{schema}{{\Phi}Update}
\Delta System
\\ \Delta File
\\ f? : ID
\where
f? \mapsto \theta~File \in fs
\\ fs' = fs \oplus \{ f? \mapsto \theta File~' \}
\end{schema}
```

and in troff as

```
.ZS \(*FUpdate
\(*DSystem
\(*DFile
f? : ID
.ZM
f? mlet theta File mem fs
fs' = fs fxov { f? mlet theta File ' }
.ZE
```

with accompanying macros defining the rendering of such mark-up.

Z's syntax and semantics have recently been standardised [ISO-Z]. In standardising the syntax of Z, a formalisation of the lexical tokens (such as the keywords, and names of definitions) used in the syntax was needed. Given the complexity of Z's orthography as noted above, and the requirements for internationalisation (the use of non-Latin character sets, such as those of Japanese and Russian) the Z standards panel took the opportunity to go one step further and define the characters from which the lexical tokens are formed. This allows the Z standard to answer such questions as *Can Z names contain superscripts and subscripts?* and *Can Z names contain multiple symbols, and mixes of symbols and letters?*, in the affirmative.

The formalisation of Z's characters has consequences for the mark-ups used in tools. These were described above as using "sequences of ASCII characters [to] encode *phrases* of a Z specification". ﬃow that Z's characters have been formalised, the "phrases" that are encoded are individual characters or sequences of characters.

This paper is a report about the work on standardising these aspects of Z, by two members of the Z standards panel. Sections 2 to 6 provide information for Z authors; sections 7 to 10 provide additional information for Z tool builders.

2 Internationalisation and Character Classes

The Z standard formalises the syntax of the Z notation, including not just the lexical tokens from which phrases are constructed but also the characters that make up these tokens. The character set includes the letters, digits and other symbols of ASCII [ISO-ASCII], and also the mathematical symbols used by Z,

and the letters of other alphabets. As few restrictions as possible are imposed on the standard character set, and a mechanism to extend the set is provided.

Other standards bodies have been working specifically on the internationalisation issue. The Universal Multiple-Octet Coded Character Set (UCS) [ISO-UCS, Unicode 2001] includes practically all known alphabets, along with many mathematical and other symbols. The Z standards panel has worked with the STIX project (who provide input on mathematical symbols for Unicode) to ensure that all of Z's standard mathematical symbols are present in UCS. Hence UCS can serve as the definitive representation of essentially any character that might be used in a Z specification.

Each character in UCS has a *code position* and attributes such as *name* and *general property*. The general property distinguishes letters, digits, and other characters, along with further distinctions such as whether a digit is decimal. The characters used in Z are partitioned into four classes: `LETTER`, `DIGIT`, `SPECIAL` and `SYMBOL`. These are referred to collectively as *Z characters*.[1]

```
ZCHAR   = LETTER | DIGIT | SPECIAL | SYMBOL ;
LETTER  = 'A' | 'B' | 'C' | ... | 'a' | 'b' | 'c' | ...
        | 'Δ' | 'Ξ' | 'θ' | ...
        | 'ℙ' | 'ℕ' | ...
        ;
DIGIT   = '0' | '1' | '2' | ... ;
SPECIAL = '′' | '?' | '!' | '(' | ')' | '[' | ... ;
SYMBOL  = '∧' | '∨' | '⇒' | '.' | ',' | ... ;
```

The characters of the `SPECIAL` class have certain special roles: some delimit neighbouring tokens, some glue parts of words together, and some encode the boxes around mathematical text. They cannot be used arbitrarily as characters within Z names. Every other character of UCS can be considered to be present in one of the `LETTER`, `DIGIT` or `SYMBOL` classes according to its UCS general property, and can be used as a character within Z names. Some of these characters are explicitly required by the Z standard; any other UCS character can be used in a Z specification in the appropriate class, though particular tools might not support them all. A UCS character's general property can be determined from the Unicode character database [Unicode 2001].

In the unlikely case of a character being needed that is not in UCS, use of that character is permitted, but its class (`LETTER`, `DIGIT` or `SYMBOL`) is not predetermined, and there is less chance of portability between tools supporting it.

Paragraph outlines are encoded as particular UCS characters that are in the `SPECIAL` class. This allows their syntax to be formalised in the usual way, independent of exactly how those outlines are rendered.

[1] The standard dialect of BNF [ISO-BNF] is used in enumerating them.

3 Tokenisation

Several requirements were considered by the Z panel in defining the internal structure of Z tokens.

1. ﬃames should be able to contain multiple symbols (such as $\frown/$ and ::), and even combinations of letters and symbols (such as $\bot_x$).
2. The user should not need to type white space between every pair of consecutive Z tokens. For example, `x+y` should be lexed as three tokens x, $+$, and y, not as a single token.
3. Subscripts and superscripts should be permitted. Single digit subscripts, such as in x_0 or $\mathbb{N}_1$ that have conventionally been used as *decorations*, and more sophisticated subscripts and superscripts should be permitted within names, as in, for example, x_+, y_{min}, or even a^{ξ_3} should the specifier so desire.

Lexical tokens are formalised in terms of the classes into which Z characters are partitioned. For example, a token that starts with a decimal digit character is a numeral, and that numeral comprises as many decimal digits as appear consecutively in the input.

```
NUMERAL = DECIMAL , { DECIMAL } ;
```

ﬃames in Z are formed from words with optional strokes, for example *current*, $next'$, $subsequent''$, *input*?, *output*!, $x0$ (a two character word with no stroke) and x_0 (the one character word x with the single stroke 0).

```
NAME = WORD , { STROKE } ;
```

Words can be alphanumeric, as in the examples of names above. Alphanumeric word parts are formed from characters of the `LETTER` and `DIGIT` classes. The first word part cannot start with a `DIGIT` character, as that would start a numeral.

```
ALPHASTR = { LETTER | DIGIT } ;
```

Words can be symbolic, as in $\frown/$. Symbolic word parts are formed from characters of the `SYMBOL` class.

```
SYMBOLSTR = { SYMBOL } ;
```

In addition, words can be formed from several parts "glued" together, allowing a controlled combination of letters and symbols. Each part of a word can be either alphanumeric or symbolic. The glue can be an underscore, for example *one_word* and x_+_y, or subscripting and superscripting motions, for example x_b, y^2, and $z_\oplus$. These motions are encoded as `WORDGLUE` characters within the `SPECIAL` class. Making those motion characters visible, the examples can be depicted as $x \searrow b \nwarrow$, $y \nearrow 2 \swarrow$, and $z \searrow \oplus \nwarrow$.

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

To support words that are purely superscripts, the parts being glued can be empty. For example $^\sim$ is a word formed from three characters: a superscripting motion glue character, a `SYMBOL`, and a motion back down again glue character, $\nearrow\sim\swarrow$. Subscripts and superscripts can be nested, for example a^{b^c} is $a \nearrow b \nearrow$

$c \swarrow \swarrow$ and x^{y_z} is $x \nearrow y \searrow z \nwarrow \swarrow$. For historical reasons, however, a subscripted digit at the end of a name, for example p_2, is lexed as a STROKE, not a subscript part of the word.

```
WORDPART = WORDGLUE , ( ALPHASTR | SYMBOLSTR ) ;

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

An empty wordpart can be regarded as an ALPHASTR or a SYMBOLSTR: the ambiguity in the formalisation is irrelevant.

The Z character SPACE separates tokens that would otherwise be lexed as a single token.

A reference to a definition whose name is decorated means something different from a decorated reference to a schema definition. Standard Z requires that these two uses of strokes be rendered differently. A STROKE that is part of a NAME is distinguished from a STROKE that decorates the components of a referenced schema by the absence of SPACE before the STROKE. (See section 3.2 in [Toyn 1998] for more detailed motivation of this.) For example, S' is the name comprising the word S and the stroke $'$, whereas $S\ '$ is the decoration expression comprising the (schema) name S and the decorating stroke $'$. A less subtle distinction of decorating strokes is to render them with parentheses—as in $(S)'$—as then the stroke cannot be considered to be part of the name.

This standard lexis for Z is a compromise between flexibility and simplicity. Apart from subscripting and superscripting motions, there is no notation concerned with rendering the characters: nothing is said about typeface, size, orientation or colour, for example. The meaning of a specification is independent of rendering choices. For example, '*dom*', 'dom' and '**dom**' are regarded as representing the same Z token. For historical reasons, however, there are three classes of exception to this rendering rule.

1. The 'doublestruck' letters, such as $\mathbb{N}$ and $\mathbb{F}$, are regarded as distinct from the upper case letters such as N and F respectively.
2. The schema operators $\setminus$, $\upharpoonright$ and $⨾$ are regarded as distinct from the toolkit operators $\setminus$, $\upharpoonright$, and $⨾$.
3. The toolkit's unary negation - is regarded as distinct from binary minus $-$.

4 Mark-ups

All of Z notation—its mathematical symbols, any other UCS character, and the paragraph outlines—can be written using a pen, pencil or chalk. But the input devices used to enter Z specifications for processing by tools are usually restricted to ASCII characters. Even those tools that offer virtual keyboards—palettes of symbols—also use ASCII encodings. So there is a need to encode phrases of

Z specifications by sequences of ASCII characters. These encodings are called *mark-ups*. For any particular Z specification, there can be many different mark-ups all of which convert to the same sequence of Z characters. For example, the LaTeX mark-ups `\dom` and `dom` both convert to the Z character sequence 'dom', but may be rendered differently by LaTeX.[2]

The Z standard defines two mark-ups:

1. the e-mail mark-up, suitable for use in ASCII-based plain text messages, and for mnemonic input to Z tools like Formaliser [Stepney];
2. LaTeX mark-up, which allows a Z specification to be embedded within a document to be typeset by LaTeX [Lamport 1994], and used as the input (in pre-standard versions of the mark-up) to Z tools like CADiZ [Toyn], Z/EVES [Saaltink 1997], *f*UZZ [Spivey 1992a], ProofPower [Arthan] and Zeta [Grieskamp].

Use of a standard mark-up allows specifications to be portable between different tools that conform to the standard in this way. The UCS representation could become another basis for portability of raw Z specifications.

Other Z tools may define their own mark-ups (for example, CADiZ also has a troff mark-up). Any particular mark-up is likely to provide the following features:

1. some means of delimiting formal Z paragraphs from informal explanatory text;
2. names for non-ASCII characters;
3. directives for extending the mark-up language to cope with the names of user-defined operators.

Precise details of conversion depend on the particular mark-up.

5 E-mail Mark-up

In the e-mail mark-up, each non-ASCII character is represented as a string visually-suggestive of the symbol, enclosed in percent signs. There are also ways of representing paragraph outlines.

This mark-up is designed primarily to be readable by people, rather than as a purely machine-based interchange format. To this end, it permits the percent signs to be left out where this will not cause confusion to the reader. However, there are no guarantees that such abbreviated text is parsable. If machine processing is required, either all the percent signs should be used, or a mark-up designed for machines, such as the LaTeX markup, should be used.

In full e-mail mark-up, the example at the beginning of this paper is

[2] Prior to the Z standard, mark-ups were usually converted directly to lexical tokens, and so these examples could be, and usually were, converted to different tokens. Now that Z characters have been formalised, it is simpler to convert mark-up to sequences of Z characters, but this means the examples will necessarily convert to the same Z token.

```
+-- %Phi%Update ---
  %Delta%System
  %Delta%File
  f? : ID
|--
  f? %|-->% %theta% File %e% fs
  fs' = fs %(+)% { f? %|-->% %theta% File ' }
---
```

In a more readable, abbreviated form, the predicate part might be written as

```
  f? |--> %theta File %e fs
  fs' = fs (+) { f? |--> %theta File ' }
```

6 LaTeX Mark-up

LaTeX mark-ups were already in widespread use before Z was standardised. The Z standard LaTeX mark-up is closely based on two of the most widely used ones [Spivey 1992a, King 1990] for backwards compatibility. The Z standard requires that the sequence of lexical tokens that is perceived by reading the rendering shall be the same as the sequence of lexical tokens that it defines from the conversion. As the rendering of LaTeX mark-up was already defined, the conversion of LaTeX mark-up has had to be defined carefully. Including changes to Z, this has resulted in the following changes to LaTeX mark-up.

1. A conjecture paragraph is written within a `zed` environment, with the mark-up of its $\vdash?$ keyword being `\vdash?`.
2. A section header is enclosed within a new environment, `zsection`, with the mark-up of its keywords being `\SECTION` and `\parents`. (This capitalisation follows the same pattern as is used with `\IF`, `\THEN`, `\ELSE` and `\LET`, where the corresponding lower-case `\LaTeXcommand`s have already been given different definitions in LaTeX.)
3. Mutually recursive free types [Toyn *et al.* 2000] are separated by the & keyword with mark-up `\&`.
4. The set of all numbers $\mathbb{A}$ is marked-up as `\arithmos`.
5. The '`%%Zchar`' mark-up directive defines the conversion of a `\LaTeXcommand` to a Z character,
   `%%Zchar \LaTeXcommand U+nnnn` or
   `%%Zchar \LaTeXcommand U-nnnnnnnn`
   where `nnnn` is four hexadecimal digits identifying the position of a character in the Basic Multilingual Plane of UCS, and `nnnnnnnn` is eight hexadecimal digits identifying a character anywhere in UCS, as in the following examples.
   `%%Zchar \nat U+2115`
   `%%Zchar \arithmos U-0001D538`
   For such `\LaTeXcommand`s that are used as operator words, mark-up directives '`%%Zprechar`', '`%%Zinchar`' and '`%%Zpostchar`' additionally include

`SPACE` before and/or after the character in the conversion of `\LaTeXcommand`, as in the following examples
`%%Zinchar \sqsubseteq U+2291`
`%%Zprechar \finset U-0001D53D`
which define conversions from `\sqsubseteq` to ‘ $\sqsubseteq$ ’ and from `\finset` to ‘$\mathbb{F}$ ’.

6. The ‘`Zword`’ directive defines the conversion of a `\LaTeXcommand` to a sequence of Z characters, themselves written in the LaTeX mark-up,
`%%Zword \LaTeXcommand Zstring`
as in the following example.
`%%Zword \natone \nat_1`
For such `\LaTeXcommands` that are used as operator words, mark-up directives ‘`Zpreword`’, ‘`Zinword`’ and ‘`Zpostword`’ additionally include `SPACE` before and/or after the converted Z characters, as in the following example
`%%Zinword \dcat \cat/`
which defines the conversion from `\dcat` to ‘ $\frown/$ ’.
7. The scope of a mark-up directive is the entire section in which it appears, excepting earlier directives (so that there can be no recursive application of the conversions), plus any sections of which its section is an ancestor.
8. The conversions of commands `\theta`, `\lambda` and `\mu` to Greek letters are defined by `%%Zprechar` directives, so that the necessary `SPACE` in expressions such as $\theta\ e$ need not be marked-up explicitly. (This automates a resolution of the backwards incompatibility reported in section 3.9 of [Toyn 1998] in the case of LaTeX mark-up.) The conversions for other Greek letters are defined by `%%Zchar` directives. The inclusion of spaces around the conversion of a `\LaTeXcommand` can be disabled by enclosing it in braces, allowing the remaining conversion to be used as part of a larger word. For example, `{\theta}`, `{\lambda}` and `{\mu}` are mark-ups for corresponding letters in longer Greek names.
9. Subscripts and superscripts can be single LaTeX tokens or be sequences of LaTeX tokens enclosed in braces, that is, `_\LaTeXtoken`, `_{\LaTeXtokens}`, `^\LaTeXtoken` and `^{\LaTeXtokens}`.
10. There is a new symbol $\ominus$ in the toolkit for set symmetric difference, marked-up as `\symdiff`.

7 Converting LaTeX Mark-up to Z Characters

The Z standard provides an abstract specification of the conversion from LaTeX mark-up to a sequence of Z characters. Some pseudo-code showing how that specification might be implemented is provided here.

7.1 The Mark-up Function

Much of the conversion of LaTeX mark-up to a sequence of Z characters is managed by the *mark-up function*, which maps individual `\LaTeXcommands` to sequences of Z characters. Mark-up directives provide the information to extend

```
%%Zchar     \\        U+000A
%%Zinchar   \also     U+000A
%%Zchar     \znewpage U+000A
%%Zchar     \,        U+0020
%%Zchar     \;        U+0020
%%Zchar     \:        U+0020
%%Zchar     \_        U+005F
%%Zchar     \{        U+007B
%%Zchar     \}        U+007D
%%Zinchar   \where    U+007C
%%Zchar     \Delta    U+0394
%%Zchar     \Xi       U+039E
%%Zprechar  \theta    U+03B8
%%Zprechar  \lambda   U+03BB
%%Zprechar  \mu       U+03BC
%%Zchar     \ldata    U+300A
%%Zchar     \rdata    U+300B
%%Zchar     \lblot    U+2989
%%Zchar     \rblot    U+298A
%%Zchar     \vdash    U+22A2
%%Zinchar   \land     U+2227
%%Zinchar   \lor      U+2228
%%Zinchar   \implies  U+21D2
%%Zinchar   \iff      U+21D4
%%Zprechar  \lnot     U+00AC
```

```
%%Zprechar \forall      U+2200
%%Zprechar \exists      U+2203
%%Zinchar  \in          U+2208
%%Zinchar  \spot        U+2981
%%Zinchar  \hide        U+29F9
%%Zinchar  \project     U+2A21
%%Zinchar  \semi        U+2A1F
%%Zinchar  \pipe        U+2A20
%%Zpreword \IF          if
%%Zinword  \THEN        then
%%Zinword  \ELSE        else
%%Zpreword \LET         let
%%Zpreword \SECTION     section
%%Zinword  \parents     parents
%%Zpreword \pre         pre
%%Zpreword \function    function
%%Zpreword \generic     generic
%%Zpreword \relation    relation
%%Zinword  \leftassoc   leftassoc
%%Zinword  \rightassoc  rightassoc
%%Zinword  \listarg     {,}{,}
%%Zinword  \varg        \_
%%Zprechar \power       U+2119
%%Zinchar  \cross       U+00D7
%%Zchar    \arithmos    U-0001D538
%%Zchar    \nat         U+2115
```

Fig. 1. Definition of the initial mark-up function

the mark-up function: the `\LaTeXcommand`, the sequence of Z characters to convert it to, and whether spaces can be converted before and/or after that sequence.

The mark-up for the Z core language can be largely defined by mark-up directives in the prelude section, as shown in Figure 1.[3] A tool may need to have the mapping from `\SECTION` to `section` built-in for the prelude's section header itself to be recognised. If a tool supports a character set larger than the minimal set required by the Z standard, such as further Greek and 'doublestrike' letters, these also can be introduced by directives in the tool's prelude section.

The mark-up for additional Z notation, such as that of the toolkit, can all be introduced by mark-up directives in the sections where that Z notation is defined. The mark-up function to be used in a particular section is the union of those of its parents extended according to its own mark-up directives.

[3] Each directive is required to be on a line by itself in a specification; in Figure 1 two columns are used to save space. The prelude section is an ancestor of every Z section, whether explicitly listed as a parent or not, so the directives of Figure 1 are in scope everywhere.

7.2 The Scanning Algorithm

Converting the LaTeX mark-up of a Z specification to a sequence of Z characters involves more work than merely mapping `\LaTeXcommands` to sequences of Z characters.

The mark-up directive corresponding to a particular use of a `\LaTeXcommand` may appear conveniently earlier in the same section, and it may alternatively appear later in the same section or in a parent section. Moreover, the sections may need to be permuted to establish the definition before use order that is necessary prior to further processing. Some preparation needs to be done to ensure that directives are recognised before their `\LaTeXcommands` are used. This preparation can be done in a separate first pass. The first pass need recognise only section headers and mark-up directives. A specification of it appears in section 8 below.

The second pass is where the LaTeX mark-up is converted to a sequence of Z characters. The conversion takes place in several phases, due to dependencies between them.

The first phase searches out the formal paragraphs, as delimited by `\begin` and `\end` commands of particular LaTeX environments (`axdef`, `schema`, `gendef`, `zed`, and `zsection`), eliding the intervening informal text. (For some applications it might be more appropriate to retain the informal text, but eliding it simplifies the following description.)

The second phase tokenises the LaTeX mark-up, consuming ASCII characters and producing Z characters. It:

- elides soft space (spaces, tabs and newlines excepting newlines at the ends of directives) and comments (but retains directives);
- converts `@` to • and `-` to `U+2212` and `~` to `SPACE`;
- inserts `SPACE` around math function characters (`+`, `−`, `*`, •, `|`) and after math punctuation characters (`;`, `,`) and around sequences of (having removed soft space within) math relation characters (`:`, `<`, `=`, `>`) if those math characters are not superscripts or subscripts or enclosed in braces, and with any following superscript or subscript appearing before the following `SPACE`;
- recognises `\LaTeXcommands` eliding soft space after them and converting `\begin{axdef}` to `AXCHAR`, `\begin{gendef}` to `AXCHAR GENCHAR`, `\begin{schdef}` to `SCHCHAR`, `\begin{zed}` to `ZEDCHAR`, `\begin{zsection}` to `ZED`, `\end{`*same*`}` to `ENDCHAR`, `\t`*digit* to `SPACE`, `\`*space* to `SPACE`, and remembering the names of other `\LaTeXcommands` and whether they were enclosed in braces;
- and retains every other ASCII character *xy* as UCS character `U+00`*xy*.

The third phase converts `\LaTeXcommands` to their expansions, consuming Z characters and producing Z characters. It:

- inserts `SPACE` before `\LaTeXcommands` that require it and were not in braces;
- applies the mark-up function to convert remaining `\LaTeXcommands` to Z characters, inserting `\` before each `{`, `}`, `^`, `_` and `\` in the conversion, and

processes the result (rejecting any `\LaTeXcommand` that is not in the domain of the mark-up function);
- inserts `SPACE` after `\LaTeXcommand`s that require it and were not in braces, postponing that `SPACE` to after any superscript or subscript;
- replaces `^\LaTeXcommand` by `^{`*str*`}` and `_\LaTeXcommand` by `_{`*str*`}`, where *str* is the conversion of `\LaTeXcommand` with `\` inserted before each `{`, `}`, `^`, `_` and `\` in the conversion, and processes the result;
- converts `^{`*str*`}` to $\nearrow$ *str* $\swarrow$ and `_{`*str*`}` to $\searrow$ *str* $\nwarrow$ and `^`*c* to $\nearrow$ *c* $\swarrow$ and `_`*c* to $\searrow$ *c* $\nwarrow$;
- inserts `GENCHAR` after `SCHCHAR` if appropriate, and `SPACE` after schema paragraph's `NAME`;
- elides braces that are not preceded by the `\` escape;
- removes all remaining `\` escapes (from `{`, `}`, `^`, `_` and `\`);
- and forwards all other Z characters unchanged.

The fourth phase manages the mark-up function, consuming Z characters and producing Z characters. It:

- recognises section headers, initialising the section's mark-up function using the mark-up functions of its parents, and forwarding its Z characters unchanged;
- recognises mark-up directives, revises the mark-up function of the current section, and elides their Z characters (rejecting any badly-formed mark-up directives and multiple mark-up directives for the same `\LaTeXcommand` in this same scope);
- and forwards all other Z characters unchanged.

These four phases naturally communicate via queues of characters. ﬃote that the third phase must not run too far ahead of the fourth phase, otherwise it might attempt to convert a `\LaTeXcommand` before the corresponding directive has caused revision of the mark-up function.

The pseudo-code for the four phases may be somewhat over-specified, that is, some of the individual conversions might work just as well in different phases. It may appear complicated, but—apart from careful placing of spaces around operators—is a fairly direct translation of LaTeX to Z characters, presented in detail to cover all special cases. It is presented in a form that has been abstracted from an implementation for which no mistakes are presently known.

8 Sections

ISO Standard Z specifies the syntax of Z as being a sequence of sections. It specifies the semantics of just those sequences of sections that are written in definition before use order. Sections may be presented to the user in a different order, perhaps for readability reasons. If the sections are presented to a tool in the same order as to the user, the tool needs to permute them into a definition before use order so that their conformance to the standard can be checked. As

extensions to the mark-up language can be defined in parent sections and used in subsequent sections, this permutation has to be done on the source mark-up, before the mark-up is converted to a sequence of Z characters.

If there are cycles in the parents relation, it will not be possible to find a definition before use order for the sections; such a specification does not conform to standard Z, and so no further processing of it is necessary.

A further issue regarding sections is where they come from: do they reside in files, and if so how are they arranged? The standard gives no guidance on this, as it inevitably varies between implementations. ffievertheless, there are likely to be considerable similarities between implementations, and so we believe it is useful to present a specification of one particular implementation.

8.1 Specification of Finding and Permuting Sections

This specification takes a filename as input, along with an environment in which that filename is interpreted, retrieves the mark-up of a Z specification from files determined by the filename and environment, and generates mark-up in which the sections are in a declaration before use order. This specification is written with the needs of the CADi$\mathbb{Z}$ toolset in mind, but aspects of it are relevant in other contexts. The specification has been typechecked by CADi$\mathbb{Z}$ [Toyn].

Introduction A specification written in standard Z [ISO-Z] comprises a sequence of *sections* [Arthan 1995], each of which has a header giving both its name and a list of the names of its parents. Its meaning includes the paragraphs of its parent sections as well as its own paragraphs.

For backwards compatibility with traditional Z [Spivey 1992b], a bare sequence of paragraphs (an *anonymous section*) is accepted as a specification comprising the sections of the prelude, the mathematical toolkit, and a section containing that sequence of paragraphs. A named section needs to include the toolkit explicitly as a parent if it makes use of those definitions.

The specification below assumes that sections are stored in files, possibly several per file. Each file is viewed as containing a sequence of paragraphs: the specification distinguishes formal paragraphs, informal paragraphs, and section headers. It needs to interpret the content of section headers, but does not need to interpret the mark-up within formal paragraphs.

Any references to parent sections that have not yet been read are presumed to be in files of the same name, and so those files are read. In each file, any formal paragraphs that are not preceded by a section header are treated as if there had been a section header whose name is that of the file and which has *standard_toolkit* as parent. This is similar to the treatment of anonymous sections in the Z standard. A file's name need not be the same as any of the sections it contains, in which case that name is useless from the point of view of finding parent sections, but it is useful as a starting point for a whole specification.

This specification makes use of the standard mathematical toolkit.

section *sortSects* parents *standard_toolkit*

Data types Strings are encoded as sequences of naturals. These naturals can be viewed as UCS code positions. Here we use CADi$\mathbb{Z}$'s non-standard string literal expressions, such as "*ropey example*", to display such strings.

$$String == \mathrm{seq}\,\mathbb{N}$$

ffiames (of both files and sections) are represented by strings. The form of names would be irrelevant to this specification but for literal names such as "*prelude*".

$$Name == String$$

Only certain kinds of paragraphs need be distinguished. Informal text between formal paragraphs is retained for possible display in the same order between the formal paragraphs. Informal and formal paragraphs are each treated as untranslated strings, but distinguished from each other to enable the detection of anonymous sections. Section headers are treated like paragraphs in this specification.

$$\begin{aligned} Paragraph ::= {} & Informal\langle\!\langle String\rangle\!\rangle \\ & \mid Formal\langle\!\langle String\rangle\!\rangle \\ & \mid SectionHeader\langle\!\langle [name : Name;\ parentSet : \mathbb{F}\,Name]\rangle\!\rangle \end{aligned}$$

The file system is modelled as a function from pathnames (formed of directory and file names) to sequences of paragraphs. This avoids having to specify the parsing of mark-up: this specification is independent of any particular mark-up, though implementations of it will be for specific mark-ups. Section headers can have been distinguished by the section keyword, if not distinguished by other mark-up.

$$Directory == Name$$

$$FileSystem == Directory \times Name \rightarrow\!\!\!\!\!\!+\ \mathrm{seq}\,Paragraph$$

Sections are represented as sequences of paragraphs in which an explicit section header begins each section.

$$\begin{aligned} & Section == \\ & \quad \{ps : \mathrm{seq}_1\,Paragraph \mid \\ & \qquad head\ ps \in \mathrm{ran}\,SectionHeader \\ & \qquad \land \mathrm{ran}(tail\ ps) \cap \mathrm{ran}\,SectionHeader = \varnothing\} \end{aligned}$$

Environment This specification operates in an environment comprising: the file system *fs*; the current working directory name *cwd*; the name of the directory containing the toolkit sections *toolkitDir*; and an environment variable

$SECTIONPATH$ giving the names of other directories from which sections may be read. The environment is modelled as the global state of the specification. Its value is not changed by the specification.

$$\begin{array}{l} fs : FileSystem \\ cwd, toolkitDir : Directory \\ SECTIONPATH : \mathrm{seq}\, Directory \end{array}$$

Functions The function $sectionToName$ is given a section and returns the name of that section. The name returned is that in the section header that is the section's first paragraph.

$$\begin{array}{l} sectionToName == \\ \quad \lambda\, s : Section \bullet ((SectionHeader^{\sim})\ (head\ \ s)).name \end{array}$$

The function $sectionsToParents$ is given a set of sections and returns the set containing the names of the parents referenced by those sections.

$$\begin{array}{l} sectionsToParents == \\ \quad \lambda\, ss : \mathbb{F}\, Section \bullet \\ \quad\quad \bigcup\{s : ss \bullet ((SectionHeader^{\sim})\ (head\ \ s)).parentSet\} \end{array}$$

The function $filenameToParas$ is given a search path of directory names and a file name and returns the sequence of paragraphs contained in the first file found with that name in the path of directories to be searched. If no file with that name is found, an empty sequence of paragraphs is returned (and an error should be reported by an implementation).

$$\begin{array}{l} filenameToParas : \mathrm{seq}\, Directory \times Name \twoheadrightarrow \mathrm{seq}\, Paragraph \\ \hline \forall\, n : Name \bullet \\ \quad filenameToParas(\langle\,\rangle\ , n) = \langle\,\rangle \\ \forall\, d : Directory;\ path : \mathrm{seq}\, Directory;\ n : Name \bullet \\ \quad filenameToParas(\langle d \rangle \frown path, n) = \\ \quad\quad \mathrm{if}(d, n) \in \mathrm{dom}\, fs \\ \quad\quad \mathrm{then}\, fs\ \ (d, n) \\ \quad\quad \mathrm{else}\, filenameToParas\ \ (path, n) \end{array}$$

The function $filenameToParagraphs$ is given a filename and returns the sequence of paragraphs contained in the first file found with that name in the path of directories to be searched. The directory of toolkits is always searched first (so that its sections cannot be overridden), then whatever directories are explicitly listed in the SECTIOﬃPATH environment variable, and finally the current working directory.

$$\begin{array}{l} filenameToParagraphs == \\ \quad \lambda\, n : Name \bullet \\ \quad\quad filenameToParas\ \ (\\ \quad\quad\quad \langle toolkitDir \rangle \frown SECTIONPATH \frown \langle cwd \rangle\ \ , n) \end{array}$$

The function *addHeader* reads the named file. If the file starts with a section header, but the name of that section differs from that of the file, an empty section is introduced to prevent re-reading of the file. If the file starts with an anonymous section, the sequence of paragraphs is prefixed with a section header. If the anonymous section has any formal paragraphs, it is named after the file, otherwise it is given a different name in case the first named section has that name.

$$
\begin{array}{l}
addHeader == \\
\quad \lambda\, n : Name \bullet \\
\quad\quad \mathrm{let}\; ps == filenameToParagraphs\;\; n \bullet \\
\quad\quad\quad (\mu\, pref, suff : \mathrm{seq}\, Paragraph \mid pref \mathbin{\frown} suff = ps \\
\quad\quad\quad\quad \land\ \mathrm{ran}\, pref \cap \mathrm{ran}\, SectionHeader = \varnothing \\
\quad\quad\quad\quad \land\ (suff = \varnothing \lor head\;\; suff \in \mathrm{ran}\, SectionHeader) \bullet \\
\quad\quad\quad\quad\quad \mathrm{if}\; pref = \varnothing\; \mathrm{then} \\
\quad\quad\quad\quad\quad\quad \mathrm{if}\; sectionToName\;\; suff = n\; \mathrm{then} \langle\, \rangle \\
\quad\quad\quad\quad\quad\quad \mathrm{else} \langle SectionHeader\;\; \lblot \\
\quad\quad\quad\quad\quad\quad\quad name == n, \\
\quad\quad\quad\quad\quad\quad\quad parentSet == \varnothing\; \rblot \rangle \\
\quad\quad\quad\quad\quad \mathrm{else\,if}\; \mathrm{ran}\, pref \cap \mathrm{ran}\, Formal \neq \varnothing\; \mathrm{then} \\
\quad\quad\quad\quad\quad\quad \langle SectionHeader\;\; \lblot \\
\quad\quad\quad\quad\quad\quad\quad name == n, \\
\quad\quad\quad\quad\quad\quad\quad parentSet == \{\text{"}standard_toolkit\text{"}\}\; \rblot \rangle \\
\quad\quad\quad\quad\quad \mathrm{else} \\
\quad\quad\quad\quad\quad\quad \langle SectionHeader\;\; \lblot \\
\quad\quad\quad\quad\quad\quad\quad name == n \mathbin{\frown} \text{"}informal\text{"}, \\
\quad\quad\quad\quad\quad\quad\quad parentSet == \varnothing\; \rblot \rangle) \\
\quad\quad\quad \mathbin{\frown} ps
\end{array}
$$

The function *filenameToSections* reads the named file and partitions its sequence of paragraphs into the corresponding sequence of sections.

$$
\begin{array}{l}
filenameToSections == \\
\quad \lambda\, n : Name \bullet \\
\quad\quad (\mu\, ss : \mathrm{seq}\, Section \mid \frown/\, ss = addHeader\;\; n)
\end{array}
$$

The function *readSpec* is given a set of names of files to be read and a set of sections already read from files. It returns the set of sections containing those already read, those read from the named files, and those read from files named as ancestors of other sections in this set. A file is read only if the named parent has not already been found in previous files and is not present anywhere in the current file; the parent section could be defined later in the current file, in which case any file with the name of the parent is not read. The sections should all have different names (otherwise an implementation should report an error); this specification merges sections that are identical.

$$
\begin{array}{l}
readSpec : \mathbb{F}\, Name \times \mathbb{F}\, Section \rightarrow \mathbb{F}\, Section \\
\hline
\forall\, ss : \mathbb{F}\, Section \bullet \\
\quad readSpec\ (\varnothing\ , ss) = ss \\
\forall\, ns : \mathbb{F}\, Name;\ ss : \mathbb{F}\, Section \bullet \\
\quad readSpec\ (ns, ss) = \\
\qquad \mu\, ss_2 == \bigcup \{ n : ns \bullet \operatorname{ran}(filenameToSections\ \ n) \} \\
\qquad\quad \mid \#ss_2 = \#(sectionToName(\!|\ ss_2\ |\!)) \bullet \\
\qquad\qquad readSpec\ ((sectionsToParents\ \ ss_2 \setminus \\
\qquad\qquad\qquad sectionToName(\!|\ ss\ |\!)) \setminus ns, \\
\qquad\qquad\quad ss \cup ss_2)
\end{array}
$$

The function *orderSections* is given a set of sections and returns those sections in a sequence ordered so that every section appears before it is referenced as a parent. The prelude section has to be forced to be first in the sequence, as it might not be explicitly listed as being a parent. The function is partial because of the possibility of cycles in the parents relation, about which an implementation should report errors.

$$
\begin{array}{l}
orderSections == \\
\quad \{ ss : \mathbb{F}\, Section;\ ss_2 : \operatorname{seq} Section \mid \\
\qquad \operatorname{ran} ss_2 = ss \\
\qquad \wedge\ sectionToName(head\ \ ss_2) = \text{"}prelude\text{"} \\
\qquad \wedge\ (\ \forall\, ss_3 : \operatorname{seq} Section \mid ss_3 \subseteq ss_2 \bullet \\
\qquad\quad \{ sectionToName(last\ \ ss_3) \} \cap \\
\qquad\qquad sectionsToParents\ (\operatorname{ran}(front\ \ ss_3)) = \varnothing\ ) \bullet \\
\quad (ss, ss_2) \}
\end{array}
$$

The function *sortSects* specifies the finding and permuting of sections. It takes the name of a file, and returns the ordered sequence of sections from that file and the files of ancestral sections.

$$
\begin{array}{l}
sortSects == \\
\quad \lambda\, n : Name \bullet orderSections\ (readSpec\ (\{n\} \cup \{\text{"}prelude\text{"}\}, \varnothing))
\end{array}
$$

Unformalised details Recognition of mark-up directives, perhaps moving them to the beginnings of their sections, has not been specified above.

The consistency of this specification (that at least one model satisfies it) has not been formally proven.

When this specification was implemented as part of the CADiℤ toolset, several additional features were needed, as follows.

CADiℤ's input mark-up can contain `quiet` and `reckless` directives, which are intended to disable/enable typesetting and typechecking respectively. These modes are noted as attributes of each paragraph, so that after permutation appropriate `quiet` and `reckless` directives can be inserted in the output.

The toolset can subsequently typeset sections of the Z specification, in the order in which their paragraphs appeared in the original mark-up. This is enabled

by the inclusion of filename and line number directives in the output generated by *sortSects*.

The typechecking tool can save the results of typechecking a section and its ancestors in a file, from which a subsequent invocation of the toolset on a larger Z specification can resume. The names of already typechecked sections are given as an additional argument to *sortSects*, which reads all sections as specified here, but then omits the already typechecked ones from its output.

9 Operator Templates

Once the sections have been permuted into definition before use order, and the mark-up has been converted to Z characters, there remains one further pass to be performed before the Z characters can be translated to lexical tokens. This third pass performs some processing of operator templates.

Operator templates are a kind of Z paragraph. For each operator (prefix, infix, postfix or bracketting name) defined in a specification, an operator template is required. This makes explicit the category (relation, function or generic), name, arity, precedence and associativity of the operator. It also indicates whether each operand is expected to be a single expression or a list of expressions, for example, as in the bracketted sequence $\langle A, B, C \rangle$.

Consider how sequence brackets are introduced in the toolkit. The operator template for sequence brackets says that it is a function in which $\langle$ and $\rangle$ are to be used as names in a unary operator whose operand is a list of expressions.

$$\text{function } (\langle \ ,, \ \rangle)$$

The definition of sequence brackets is in terms of seq, which results in the elements of the bracketted sequence being indexed from 1.

$$\langle \ ,, \ \rangle[X] == \lambda\, s : \operatorname{seq} X \bullet s$$

An introduction to operator templates was given in [Toyn 1998]. Standard Z has evolved since then: the precedences of prefix and postfix operators are no longer explicitly chosen but instead are fixed such that all prefix user-defined operators have higher precedence than all infix user-defined operators, and all postfix user-defined operators have higher precedence than all prefix user-defined operators.[4]

[4] This change avoids some awkward cases such as A *infix*3 *prefix*1 B *infix*2 C and A *infix*2 B *postfix*1 *infix*3 C (where the numbers indicate the desired precedence). Should these be parsed as (A *infix*3 (*prefix*1 B)) *infix*2 C and A *infix*2 ((B *postfix*1) *infix*3 C), or as A *infix*3 (*prefix*1 (B *infix*2 C)) and ((A *infix*2 B) *postfix*1) *infix*3 C, or should both be rejected? Existing Z tools gave parses that paired up differently. Other languages offering the same facility (Prolog and Twelf) disagree on which pair of parses is right. Standard Z avoids the problem by requiring high prefix and postfix precedences. This restricted notation is analogous to that of Haskell.

The names within an operator, such as $\langle$ and $\rangle$ in the above example, are assigned to particular token classes to enable parsing of operator names and operator applications. The appropriate token classes are determined from the operator template. There are separate token classes according to whether the name is the first, middle or last name in an operator, whether operands precede and follow it, and whether any preceding operand is a list of expressions. In the example, $\langle$ is assigned to the token class `L` (standing for leftmost name), and $\rangle$ is assigned to the token class `SR` (standing for rightmost name preceded by a sequence of expressions).

A name may be used within several different operators, so long as it is assigned to the same token class by all of their templates. For example, here is a specification of sequences indexed from 0.

$$\text{function} \ (\langle^0 \ \ ,, \ \ \rangle)$$

$$\langle^0 \ \ ,, \ \ \rangle[X] == \lambda\, s : \operatorname{seq} X \bullet \lambda\, n : 0 \mathrel{..} \#\operatorname{dom} s - 1 \bullet s(n+1)$$

This reuse of $\rangle$ is permitted because it is assigned to the same `SR` token class by both operator templates. Reuse of $\rangle$ as an infix, as in $_ \rangle _$, would attempt to associate it with a different token class, and so would not be permitted.

Two problems must be solved before the sequence of Z characters can be lexed. First, users may defer presenting operator templates until after the corresponding operator has been defined, yet information from an operator's template is needed to parse the operator's definition. Second, the reuse of names within different operators means that the syntax of operator templates is expressed in terms of the very token classes that operator templates establish.

The solution to the first problem is to permute operator template paragraphs to the beginnings of their sections. The solution to the second problem is to introduce, before an operator template, directives to the lexer to associate its names with token classes. These transformations can be done on the sequence of Z characters, hence the third pass. (In other words, there is no need to have separate implementations of these transformations for each mark-up.)

10 Summary

The processing of mark-up prior to lexing is summarised in Figure 2.

The first pass, section, gathers the mark-up of the specification's sections together, permutes them into a definition before use order, and brings mark-up directives to the starts of their sections.

The second pass, LaTeXmark-up, extracts the formal text, tokenises that LaTeX mark-up, expands `\LaTeXcommands` according their mark-up directives, and manages the scopes of those mark-up directives.

The third pass, optemp, brings operator templates to the starts of their sections, and generates lexis directives before them.

The resulting sequence of Z characters is ready to be lexed.

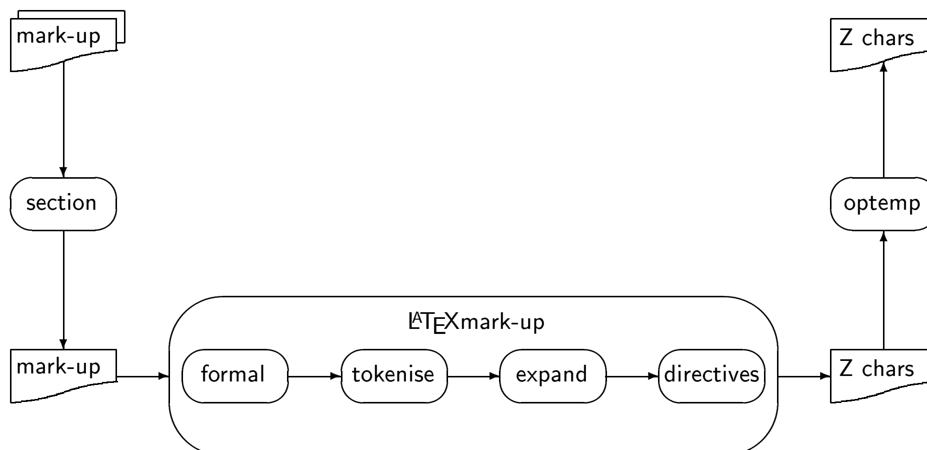

Fig. 2. Processing of mark-up prior to lexing

11 Conclusions

ISO Standard Z has answered the question "What's in a name?" in the Z context. The answer addresses the internationalisation issue. It establishes a new representation for a Z specification: its sequence of Z characters. This representation is intermediate between mark-up and lexical tokens. It simplifies the task of designing a new mark-up for Z, which can be to sequences of Z characters rather than directly to lexical tokens.

The liberal scope rules of sections and operator templates require extra processing of a Z specification prior to lexing. The paper has made explicit an order in which that processing can be done.

Acknowledgements

We thank the members of the Z standards panel for their influence, Sam Valentine for advising on the use of Z in the *sortSects* specification, the STIX committee for arranging the inclusion of Z characters in UCS, and Steve King and anonymous referees for comments on earlier draft versions of this paper.

References

[Arthan]
R.D. Arthan. The ProofPower web pages.
http://www.lemma-one.com/ProofPower/index/index.html.

[Arthan 1995]
R.D. Arthan. Modularity for Z. http://www.lemma-one.com/zstan_docs/wrk059.ps, September 1995.

[Ciancarini *et al.* 1998]
P. Ciancarini, C. Mascolo, and F. Vitali. Visualizing Z notation in HTML documents. In *ZUM'98: The Z Formal Specification Notation*, LNCS 1493, pages 81–95. Springer, 1998.

[Germán *et al.* 1994]
D.M. Germán, D. Cowan, and A. Ryman. Comments on the Z Interchange Format of the Z Base Standard Version 1.0.
ftp://ftp.comlab.ox.ac.uk/pub/Zforum/ZSTAN/papers/z-160.ps.gz, 1994.

[Grieskamp]
W. Grieskamp. The Zeta web pages. http://uebb.cs.tu-berlin.de/zeta/.

[ISO-ASCII]
ISO/IEC 646:1991. *Information Technology—ISO 7-bit Coded Character Set for Information Interchange (3rd edition).*

[ISO-BNF]
ISO/IEC 14977:1996(E). *Information Technology—Syntactic Metalanguage—Extended BNF.*

[ISO-UCS]
ISO/IEC 10646:2000. *Information Technology—Universal Multiple-Octet Coded Character Set (UCS).*

[ISO-Z]
ISO/IEC 13568:2001. *Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics: Draft International Standard.* To be published.

[King 1990]
P. King. Printing Z and Object-Z LaTeX documents. University of Queensland, 1990.

[Lamport 1994]
L. Lamport. *LaTeX: A Document Preparation System—User's Guide and Reference Manual, 2nd edition.* Addison-Wesley, 1994.

[Saaltink 1997]
M. Saaltink. The Z/EVES system. In *ZUM'97: The Z Formal Specification Notation*, LNCS 1212, pages 72–85. Springer, 1997.

[Spivey 1992a]
J.M. Spivey. The *f*UZZ manual, 2nd edition. Computer Science Consultancy, 1992.

[Spivey 1992b]
J.M. Spivey. *The Z Notation: A Reference Manual, 2nd edition.* Prentice Hall, 1992.

[Stepney]
S. Stepney. Formaliser Home Page. http://public.logica.com/~formaliser/.

[Toyn]
I. Toyn. CADiZ web pages. http://www-users.cs.york.ac.uk/~ian/cadiz/.

[Toyn *et al.* 2000]
I. Toyn, S.H. Valentine, and D.A. Duffy. On mutually recursive free types in Z. In *ZB2000: International Conference of B and Z Users*, LNCS 1878, pages 59–74. Springer, 2000.

[Toyn 1998]
I. Toyn. Innovations in the notation of Standard Z. In *ZUM'98: The Z Formal Specification Notation*, LNCS 1493, pages 193–213. Springer, September 1998.

[Unicode 2001]
The Unicode Consortium. *The Unicode Standard, Version 3.1*, May 2001. http://www.unicode.org/.