Accumulated Defect Reports 13568/001..013

ISO/JTC1/SC22/WG19
Z Project Editor: *Ian Toyn*
ian@cs.york.ac.uk

September 1, 2006

Abstract

This document addresses the known defects in ISO Standard Z, proposing alternative solutions to those defects, and identifying preferred solutions. The implied changes to the standard are given in a separate document of technical corrigenda. Each top-level section corresponds to a separate defect report. The last section concerns editorial defects, while all other sections concern technical defects.

Contents

1 Use of UCS

This defect concerns errors in clause 6 and hence in annex A.

1.1 Nature of defect

UCS provides alternative encodings of similar looking characters for different purposes, from which some inappropriate encodings were chosen for the Z standard. Also, character properties could be more precisely defined via Unicode General Categories. The bibliography should refer to supporting Unicode documents.

1.2 Proposed solution

A proposed solution was offered by the Swedish comments on the standard. The technical corrigendum is based on those comments, taking into account more recent updates to Unicode. The changes have been tested in CADiℤ, with one deviation: the tool regards all characters not in other classes as being in the `SYMBOL` class, because I don't yet know of an efficient way for the tool to exclude the characters that should be excluded.

2 LaTeX zed environment

This defect concerns an omission from A.2.7.

2.1 Nature of defect

The `\begin{zed} ... \end{zed}` LaTeX environment is illustrated in the Tutorial annex, but is not introduced in any normative clause.

2.2 Proposed solution

Introduce the `zed` environment in A.2.7, and tidy some neighbouring wording.

3 LaTeX theorem environment

This defect concerns an omission from A.2.7.

3.1 Nature of defect

The LaTeX mark-up of conjecture paragraphs is not specified.

3.2 Proposed solutions

The `zed` environment

My recollection of the consensus of the panel at the time the standard was published (with the omission) was that the `\begin{zed} ... \end{zed}` LaTeX environment would be used for conjectures just as it is used for other unboxed paragraphs.

The `theorem` environment

The practice in existing tools is to use the `\begin{theorem} ... \end{theorem}` LaTeX environment.

One reason for tools retaining this deviation from the standard arises from their support for a non-standard notation that allows more than one unboxed paragraph within a single `zed` environment. The purpose of this notation is to reduce the vertical space consumed by the rendering. The following example illustrates an ambiguity that arises if conjectures are allowed within such an environment.

```
\begin{zed}
\vdash? true \\
x == 42
\end{zed}
```

The newline is intended to serve as paragraph separation, but where a conjecture is followed by another paragraph, the newline could also mean predicate conjunction: in the example, the `x` is conjoined onto `true`. Other unboxed paragraphs do not give rise to such ambiguities; only conjectures do. Requiring conjectures to appear in a different environment avoids the ambiguities in the LaTeX mark-up, and keeps disambiguating vertical space in the rendering. Further ambiguity exists in the corresponding UCS stream, which could be resolved by use of a different box token for conjectures, and hence a further deviation from the standard.

A second reason for not using the `zed` environment is to accept an argument specifying a name for the conjecture, as follows.

```
\begin{theorem}{name}
...
\end{theorem}
```

Preferred solution

The `theorem` environment is the preferred solution, because tools are not going to conform to the standard otherwise.

The extensions to multiple unboxed paragraphs in the same environment, and the introduction of a different box token for conjectures, are not addressed by the technical corrigendum. Nor does it add a section on conjectures to the Tutorial annex, which might also be a good thing to do. It assumes that A.2.7.6 has already been introduced by correcting the preceding defect.

4 Semantic equation for schema universal quantification expression

This defect concerns an error in 15.2.5.13.

4.1 Nature of defect

The semantic equation for schema universal quantification $\forall e_1 \bullet e_2$ does not give the intended semantic value in the case when e_1 has signature variables that are not also signature variables of e_2.

This case was not allowed in earlier versions of Z, but it is intended to be allowed in ISO Standard Z to make the rules for schema quantification uniform with those for predicate quantification. Now that the two forms of quantification can appear in the same contexts (rather than only in distinct paragraphs), this uniformity is necessary. It makes the semantics of a quantified formula that could be a schema or a predicate be the same whichever way it is read.

4.2 Proposed solution

The semantic equation should be changed to ignore the extra variables.

5 Range of operator precedences

This defect concerns the addition of a clarification in 8.3.

5.1 Nature of defect

The range of operator precedences is not constrained by the standard: it merely gives the syntax as `NUMERAL`. It is not clear from this what the intended range is.

5.2 Proposed solution

My recollection of the consensus of the panel is that the range should not be constrained, i.e. any natural number can be used. A NOTE should be added to make this explicit.

(Aside: Both CADiZ and CZT currently constrain the range to 32-bit unsigned numbers, and ProofPower limits it to a quarter of that, with no complaints that this is too constrained.)

6 Lexis of punctuation characters

This defect concerns an error in clause 7.

6.1 Nature of defect

Users have reported cases where some punctuation characters are lexed as parts of neighbouring words, rather than as the separators that they were expecting. Here are some of those cases.

- The Z phrase $(\langle x \rangle,\ \langle y \rangle)$ is perceived as a pair of sequences, but both the '$\rangle$' and ',' characters are in the SYMBOL class, and so the Lexis consumes $\rangle$, as a single WORD.
- The Z phrase $a\_, b$ is perceived as a list of names, but the '_' character is in the WORDGLUE class, and so the Lexis consumes $a\_$, as a single WORD.
- The Z phrase a_m, b is perceived as a list of names, but the subscript involves a '$\nwarrow$' character that is in the WORDGLUE class, and so the Lexis consumes a_m, as a single WORD.
- The Z phrase $Incl_2$; $Incl$ is perceived as a list of declarations, but the subscript involves a '$\nwarrow$' character that is in the WORDGLUE class, and so the Lexis consumes $Incl_2$; as a single WORD.
- The Z phrase $v_0 : T$ is perceived as a declaration, but the subscript involves a '$\nwarrow$' character that is in the WORDGLUE class, and so the Lexis consumes v_0: as a single WORD.
- The Z phrase $(\!|\ \$ == 42\ |\!).\$$ is perceived as a binding selection expression, but both the '.' and '$' characters are in the SYMBOL class, and so the Lexis consumes .$ as a single WORD. (The '$|\!)$' character is in the BRACKET class.)

The Lexis defines how a sequence of UCS characters is divided into tokens, based on a classification of their uses as Z characters. The resulting division into tokens should correspond to how an experienced Z user perceives the same text. The defect is a failure of this correspondence, leading to the Z standard being regarded as erroneous.

The expectations of an experienced Z user regarding punctuation characters can be summarised as follows. There is usually no space before ',', ';' or '.'. There is usually no space after '.' in a binding selection. There is usually space before and after a ':' in a declaration, but it might be nice to be able to omit these spaces.[1] These expectations correspond to the default rendering produced by LaTeX.

6.2 Proposed solutions

Several alternative suggestions are proposed to solve this defect.

Require extra SPACE characters

If punctuation were required to always be preceded by SPACE characters, then the Lexis would work as it is. This means changing the rendering, i.e. changing users' expectations. It effectively asserts that the Z standard is correct as it is, and that the users' expectations are wrong. However, there is too much history behind those expectations, and so this suggestion seems unacceptable.

Classify as SPECIALs

If the classifications of ',', ';', ':' and '.' were changed from SYMBOL to SPECIAL, then the Lexer would consume them as separate tokens. However, the characters could not then be used in larger WORDs. In particular, the WORDs '::=' and ',,' of the core language and the WORD '..' of the toolkit could no longer be lexed. Users might not mind ',,' being changed to avoid this, but surely the others need to be recognised without change.

[1] John Wordsworth's "Software Development in Z" omits space before ':'.

Use distinct UCS characters

If the classifications of ‘,’, ‘;’, ‘:’ and ‘.’ were changed from `SYMBOL` to `SPECIAL`, as in the previous suggestion, and similar-looking but distinct UCS characters were used when they are wanted in longer `WORD`s, then the Lexis would consume them as separate tokens yet larger `WORD`s could apparently make use of them. Unfortunately, if different characters are rendered the same, then they will be perceived as being the same, i.e. mis-lexed by the user. Although users are not prohibited from using UCS characters that look the same as others, they should not be encouraged to do so. Indeed, every effort should be made to discourage users from doing that. This suggestion does the opposite.

(Aside: 6.4.7 says “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.”. This expectation of perceptions seems at odds with this proposed solution.)

Separate at ends of `WORD`s

Having consumed a `WORD`, the Lexer could remove any ‘,’, ‘;’ and ‘.’ characters from the ends to form a separate `WORD`. If there are several of these characters on the ends (not necessarily the same characters), all would be removed. This would be analogous to the lexing of subscript digits: ‘x_2’ is first lexed as a word, then the ‘$_2$’ is separated from it to give a `DECORWORD` in which ‘x’ is the word and ‘$_2$’ is a stroke. For example, ‘⟩,’ would be first lexed as a `WORD`, then the ‘,’ would be separated from it to give two `WORD`s. If a word is made up entirely of these characters, then only one `WORD` remains. Instead of changing the character classifications in advance of lexing, this suggestion looks for particular `SYMBOL`s after having lexed a `WORD`.

The needed `WORD`s ‘,,’ and ‘..’ are made up entirely of separable characters, and so would be successfully lexed as `WORD`s. In order to lex the needed `WORD` ‘::=’, the ‘:’ character is not regarded as a separable character by this suggestion. Hence spaces would still be needed around ‘:’ in declarations.

This suggestion would allow the punctuation characters to be used in larger `WORD`s, but is relatively complicated to explain and comprehend: some uses of punctuation would be separated while others aren’t. For example, the ‘,’ character in the perceived pair of sequences ‘(⟨x⟩,⟨y⟩)’ would be lexed as being in the middle of the `WORD` ‘⟩,⟨’.

Classify as `PUNCT`uation

If a new classification of Z character were introduced for ‘,’, ‘;’, ‘:’ and ‘.’ that allowed their use in `WORD`s only with themselves, then the rule for what `WORD`s can contain punctuation would be simple. These Z characters would move from the `SYMBOL` class to a new `PUNCT` class.

```
PUNCT  =  ',' | '; ' | ':' | '.' ;
```

The Lexer would have a new alternative for `WORD` to accept a sequence of one or more `PUNCT` characters.

```
  |  PUNCT , { PUNCT }
```

This involves a compromise on what larger `WORD`s can use these characters, while still allowing some user-defined names to use them. Unfortunately, it doesn’t allow the needed `WORD` ‘::=’. Suggested solutions and work-arounds for this problem include the following.

- Use UCS character U+2A74 for the '::=' `WORD`, since it looks the same. Unfortunately, this very similarity makes it likely to be mistaken for the three separate characters. Using this would violate the principle mentioned earlier that use of UCS characters that could be mistaken for others should be avoided.
- Use a completely different symbol instead of '::='. Although potentially clear, this would be unacceptable for historical reasons.
- Add a special case as another alternative production for `WORD`.

  ```
  |  ':' , ':' , '='
  ```

  Having consumed '::', the Lexer could look-ahead one character; if that character is a '=', then the `WORD` would be '::='. The character after that would not be part of this token, whatever it is.
- Generalising from the last suggestion, allow a '=' character on the end of any punctuation word, so only a single alternative production for `WORD` would be required.

  ```
  |  PUNCT , { PUNCT } , [ '=' ]
  ```

  This would enable words such as ':=' too.

Note that this proposal is to allow `PUNCT`s in `WORD`s. Allowing them one level lower, in `WORDPART`s, would not fix the problem. Allowing them one level higher, in `DECORWORD` instead, would fail to allow references to decorated instances. (Decorated instances arise from defining say ... inside a schema and then decorating the schema with ′ to give ...′.)

Recommendation

The suggestions to separate at ends of `WORD`s and classify as `PUNCT`uation (with optional '=' after punctuation) have both been implemented (at different times) in CADiℤ. Although it excludes more `WORD`s, the latter is easier to explain and gives the desired behaviour in more cases, based on typechecking (and hence lexing) hundreds of CADiℤ's test cases. In looking at the differential output from regression testing, I noticed only two `WORD`s that can no longer be lexed, namely ':|' (for non-interleaved processes) and ':|:' (for disjoint non-interleaved processes), neither of which seem to be showstoppers. Hence the classify as `PUNCT`uation (with optional '=' after punctuation) suggestion is recommended.

7 Newlines in paragraph headers

This defect concerns an unimplementable requirement in 7.5.

7.1 Nature of defect

The requirement "All newlines are soft outside of a `DeclPart` or a `Predicate`" in 7.5 is not implementable: a lexer may not know whether it is inside or outside of a `DeclPart` or a `Predicate`.

A reminder of NL, NLCHAR and \\

Newlines can have semantic significance - conjoining predicates or merging declarations - or be used merely to layout large formulae. In LaTeX mark-up, the \\ command is converted to a NLCHAR character. The lexis (7.5) classifies each NLCHAR as soft or hard, generating a NL token for a hard newline, but not for a soft newline. The concrete syntax accepts NL tokens in only certain places. The NL tokens it accepts include those that have semantic significance, i.e. those for merging declarations and conjoining predicates. The current concrete syntax does not accept NL tokens anywhere else. It is possible for phrases to contain newlines that the lexis does not classify as soft and that the concrete syntax does not accept; such phrases are not given meanings by the standard, and parsers should reject them as syntactically erroneous. Such erroneous phrases are rare: newlines are allowed for layout purposes almost anywhere.

An example

Here is an example illustrating soft newlines, accepted hard newlines, and rejected hard newlines.

```
\begin{schema}{Sch}[X]
Incl \\ ’ \\
a : X
\where
a = \\
incl’
\end{schema}
```

The newlines that layout this LaTeX mark-up over seven lines are irrelevant. It is the three occurrences of the \\ command that give rise to three NLCHARs in the conversion of the above mark-up to this UCS...

```
SCHCHAR GENCHAR Sch [X] Incl NLCHAR ’ NLCHAR a : X | a = NLCHAR incl’ ENDCHAR
```

(Each word ending CHAR should be read as a single character.) The third NLCHAR follows an infix symbol (=), so is within a formula that inevitably extends onto the next line, and so this NLCHAR is classed as soft. The first and second NLCHARs are between characters that give rise to nofix tokens (STROKE is not regarded as postfix); these NLCHARs are classified as hard, and NL tokens are generated in the stream of tokens that is input to the concrete syntax...

```
    GENSCH Sch [X] Incl NL ’ NL a : X | a = incl’ END
```

(The spellings are shown here rather than NAME and operator tokens.) The second NL token serves to merge declarations, so this one is accepted by the concrete syntax. However, the first NL token is not accepted, and a parser should report a syntax error there.

The problem

Suppose the example is modified so that the stream of characters that is input to the lexis has an extra NLCHAR at the end of the paragraph header...

```
SCHCHAR GENCHAR Sch [X] NLCHAR Incl ...
```

This UCS stream can never arise from the conversion of LaTeX mark-up, as for example is currently done by CADiZ, but it can arise if the stream is prepared directly using a UCS editor such as the jEdit tool used in CZT. In the jEdit editor, it is desirable to have a `NLCHAR` after a paragraph header, so that the layout is as one would expect. In the current standard, 7.5 has the requirement "All newlines are soft outside of a `DeclPart` or a `Predicate`", hence the lexis classifies this `NLCHAR` as soft, and a parser should accept this input. Unfortunately, this requirement is a problem for builders of lexers, as a lexer cannot tell whether it is processing outside of a syntactic context such as a `DeclPart` or a `Predicate`. Tool builders can find a work-around that still conforms to the current standard, but the interface between lexer and parser deviates from the interface between lexis and concrete syntax. More generally, the difficulty illustrated by this example arises with `NLCHAR`s that are at the end of the headers of schema definition, generic axiomatic, and generic schema definition paragraphs. (`NLCHAR`s at the ends of headers of other paragraphs are easily classified as soft based on neighbouring tokens.)

The syntactically-oriented classification requirement turns out to be clearly wrong for some other newlines. For example,

```
ZEDCHAR i1 == i2 NLCHAR i3 ENDCHAR
```

The `NLCHAR` in this horizontal definition paragraph has semantic significance: it could be conjoining schemas `i2` and `i3` (the type inference rules decide that later). So the lexis should classify this `NLCHAR` as hard. Although the neighbouring tokens classification rules suggest it is hard, those are overridden by the requirement "All newlines are soft outside of a `DeclPart` or a `Predicate`". This requirement was written with an abstract sense of `DeclPart` in mind, which would have covered this `==` definition, but the concrete syntax for this `Paragraph` refers to `Expression` not `DeclPart`. More generally, the problem illustrated by this example occurs for all the `Paragraph` productions of the concrete syntax in which `Expression` is used. This requires a fix, and we might as well avoid the earlier difficulty at the same time.

7.2 Proposed solutions

Accept necessary newlines in paragraph headers

Suppose we drop the syntactically-oriented classification requirement "All newlines are soft outside of a `DeclPart` or a `Predicate`". `NLCHAR`s at the end of paragraph headers would then give rise to `NL` tokens only for the three paragraph productions listed earlier. The concrete syntax can be changed to accept `NL` tokens in those three productions. Semantics then needs to be specified for phrases using such `NL` tokens, which is easily done using syntactic transformation rules to elide those tokens. The acceptance of these `NL` tokens by the concrete syntax should be optional: mark-up converters might have as much trouble in generating them as a lexer has in removing them. This optionality also has the advantage that the annotated syntax (no `NL`s) remains a subset of the concrete syntax (optional `NL`s) in this respect. Would any other previously soft `NLCHAR`s become hard once that requirement were dropped? Yes - just those between a schema name and a generic parameter list, as in this example.

```
SCHCHAR GENCHAR Sch NLCHAR [X] ...
```

Laying out the generic parameters beneath the schema name resembles a schema inclusion, though formally the `GENCHAR` distinguishes those. Sometimes generic parameter lists are long enough to need to be split across multiple lines. That can be done at commas within the list, or inside the brackets. So there doesn't seem to be any need to accept a newline before the whole list. This proposal treats it as a rejected hard newline.

The effects of this proposal are as follows. The proposal fixes (by removal) the broken requirement in 7.5. The changes to the soft/hard classification within the standard make it more closely reflect what existing tools do. This is more helpful to future tool builders. The Z language is changed to exclude newlines between schema names and generic parameter lists.

Accept all newlines in paragraph headers

A variation on the above proposal would be, for uniformity, to add optional `NL` tokens to all paragraph header productions in the concrete syntax, not just those three, and hence the box tokens would not need to be in newline category `BOTH`. However, this would make the concrete syntax longer, would require more syntactic transformation rules, and more tokens would be passed between the lexer and parser in tools, without having any effect on the Z language.

Recommendation

The first, less uniform but more efficient, solution is recommended.

8 Look-ahead in lexer

This defect concerns an over-specific clarification in 8.4.

8.1 Nature of defect

In 8.4, the disambiguation of $i\ [e_1;\ e_2]$ (the application of a name to a schema construction expression) from $i[e_1]$ (the instantiation of a name by a single expression) is discussed as an example of a case where look-ahead is needed within a lexer. It has been demonstrated that these can be disambiguated by an LALR(1) parser without any need for distinct [tokens, and hence there is no need for look-ahead in the lexer.

For those interested and familiar with grammars based on mine, the alternative is to extend the *InnerForm* grammar rule with the following new production.

$$InnerForm = InnerForm\ '['\ SchemaTextNotExpr\ ']'\ OptAppendages$$

$$\begin{aligned} OptAppendages = {} & Appendages \\ | \quad & \qquad\qquad -- \ empty \\ ; & \end{aligned}$$

$$\begin{aligned} Appendages = {} & Appendages\ \ Appendage \\ | \quad & Appendage \\ ; & \end{aligned}$$

Appendage = *STROKE*
| '[' *RenameList* ']'
;

Some fiddling with productions' precedences may also be needed for this to work.

8.2 Proposed solution

Omit the inappropriate example from 8.4.

9 Use of meta-language *decor* operator

This defect concerns an error in Table 11 of 4.2.6.

9.1 Nature of defect

The *decor* definition in Table 11 is for one or more strokes, yet the definition is used for possibly zero strokes, e.g. by the type inference rule for binding construction expressions in 13.2.6.9.

9.2 Proposed solution

Change the definition to admit use with zero strokes.

10 Operators in generic horizontal definitions

This defect concerns an omission in 8.2.

10.1 Nature of defect

The syntax for generic horizontal definitions has an unnecessary restriction that prevents the definition of operators.

10.2 Proposed solution

Change the syntax to allow an operator in a generic horizontal definition.

11 LaTeX rendering of multiple strokes

This defect concerns an omission in annex A.

11.1 Nature of defect

The inappropriateness of the obvious LaTeX mark-ups for multiple strokes, such as '`x_1''` and '`x'_1`', is not deprecated, as it should be since they are rendered identically as x_1' despite having different conversions.

11.2 Proposed solution

Add a requirement that this mark-up be deprecated, and add an example of the preferred, less obvious, mark-ups '`x_1{}''` and '`x'{}_1`', which render as $x_1{}'$ and x'_1.

12 Informal text

This defect concerns the additions of clarifications in 5.2.2, 6.1 and A.1.

12.1 Nature of defect

The processing of informal text in specifications is not defined: clause 6 assumes that informal text has been eliminated, yet nothing eliminates informal text, not even annex A.

12.2 Proposed solution

Make the assumption explicit in clause 6. Clarify that processing of mark-up shall eliminate informal text. How that is done depends on the particular mark-up, so don't define it.

13 Miscellaneous editorial corrections

This defect concerns various clarifications spread throughout the standard.

13.1 Nature of defect

These changes address editorial mistakes and clarifications that are thought not to need any justification, hence there is no discussion of proposed solutions.

Acknowledgements

Lots of people have helped in the evolution of the Technical Corrigenda. In alphabetical order, and avoiding attributing defect reports to particular people, these include Rob Arthan, Leo Freitas, Kent Karlsson, Petra Malik, Andrew Martin, Tim Miller, Fiona Polack, Roger Scowen, Sam Valentine, Susan Stepney and Mark Utting.