ZTC: A Type Checker for Z Notation

User's Guide

Version 2.2, October 2002

Xiaoping Jia
School of Computer Science, Telecommunication, and Information Systems
DePaul University
Chicago, Illinois, U.S.A.

Contents

Acknowledgments

Many people have contributed to the development and improvement of ZTC. Particularly, I wish to thank Mr. Philip Garofalo for developing the first ZSL parser, Mr. Sotirios Skevoulis for spending countless hours testing the tool, Mr. Andrej Semrl and Marc Mengle for their insightful comments regarding ZTC. I also wish to thank all my students in the *Formal Methods* class (SE 431) for suffering through various β-versions of the tool. Their enthusiasm about the tool and formal methods has been a great source of encouragement for me to complete the work.

1 Introduction

The Z notation[1] is a model-oriented formal specification language developed by the Programming Research Group at Oxford University Computing Laboratory in the early 80's. Since then, Z has been used to specify a wide spectrum of software systems including database systems, transaction systems, distributed computing systems, and operating systems [2]. The most notable success of Z is the specification of CICS Application Programming Interface (API) by IBM United Kingdom Laboratories at Hursley Park [3]. Approximately 37,000 lines of code were produced from Z specifications and designs, and it was reported that the code has approximately 2.5 times fewer problems than the code that was not specified in Z.

This document is not intended to be a tutorial of the Z notation and formal specifications. It assumes that you are familiar with the Z notation. If you are not familiar with Z, there several sources to learn about Z:

- *An Introduction to Z and Formal Specification* [4] by Mike Spivey gives a brief introduction to Z and model-oriented formal specifications.
- *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering* [5] by J.B. Wordsworth and *Z: An Introduction to Formal Methods* [6] by A. Diller are introductory textbooks on formal methods and Z.
- The internet newsgroup `comp.specification.z` and the World-Wide-Web site `http://www.comlab.ox.ac.uk/archive/z.html` provide many pointers to recent development and materials in the field.

Z is a non-executable but strongly-typed specification language. ZTC is a type-checker for Z, which determines if there are syntactical and typing errors in Z specifications. There is no compiler for Z. However, there are tools to animate, or execute, subsets of Z[1].

ZTC accepts two forms of input: LaTeX with `oz` or `zed` packages, and ZSL. `Oz`[7] and `zed`[8] are LaTeX packages (style options) developed Paul King and Mike Spivey, respectively, for typesetting Z in LaTeX. ZSL is an ASCII version of Z designed by the author. ZSL is welcome by students and newcomers of Z who are not familiar with LaTeX, so they can write and type-check Z specifications without the extra hurdle of learning LaTeX. Unlike the SGML based Z interchangeable format proposed by the Z Standard Committee, which is primarily intended for tools not human readers, ZSL is designed to be readable and try to retain the visual appearance of Z specifications as much as possible. ZSL is also useful for ASCII based electronic communications, such as e-mail, involving Z specifications. ZTC can perform translations between LaTeX and ZSL. A brief description and examples of the two input forms are given in Section 2.

If you choose to use the LaTeX input form but you should first get familiar with with `oz` [7] or `zed` [8]. Then, read Section 3. If you choose to use ZSL, you can safely skip Section 3, read Section 4, Section 5 will discuss many advanced and new features in version 2.0. Most of them applies to both input forms. Use Appendix A as a reference for both input forms.

[1]ZANS is an experimental tool developed here at the Software Engineering Laboratory, DePaul University, that animates a subset of Z. ZANS is also freely available and its input format is compatible with that of ZTC. A research project funded by the National Science Foundation to study and develop tools for design refinement and code synthesis based on Z is also underway here.

This user's guide describes ZTC version 2.1. ZTC is now available on the following platforms:

- Microsoft Windows 9x, NT, 2000, and XP, and
- Linux

2 The Input Forms

ZTC accepts tow input forms: LaTeX and ZSL. Semantically, ZSL is as expressive as the LaTeX form. However, the LaTeX form is more expressive *visually*. ZTC can translate a Z specification written in ZSL to an equivalent one in LaTeX, and *vice versa*. Using the LaTeX form will allow you to fine tune the visual appearance of the specifications, and to take advantage of some features specially designed for LaTeX (see sections 5.3 and 5.6.)

The following is a segment of a Z specification:

$$[NAME, INFO]$$

DataDictionary
$dict : NAME \pfun INFO$
$defined : \mathbb{P}\, NAME$

$defined = \mathrm{dom}\, dict$

ZTC accepts most of the LaTeX source files as is. However, some rules must be observed in order to type-check the input files. These rules are discussed in Section 3. The LaTeX input file for the above specification is given below:

```
\begin{spec}

\begin{zed}
    [ NAME, INFO ]
\end{zed}

\begin{schema}{DataDictionary}
    dict: NAME \pfun INFO \\
    defined: \power NAME
\where
    defined = \dom dict
\end{schema}

\end{spec}
```

ZSL has two style options: the *plain text* style and the *box* style. Using the ZSL plain text style, the Z specification above will be written as follows:

```
specification

[ NAME, INFO ]

schema DataDictionary
  dict    : NAME +-> INFO;
  defined : P NAME
where
  defined = dom dict
end schema

end specification
```

The box style of ZSL gives a graphical look to schema and generic boxes that resembles the original Z style. The following is the same specification in ZSL box style:

```
specification

[ NAME, INFO ]

--- DataDictionary ---------------------
|  dict    : NAME +-> INFO;
|  defined : P NAME
|----------
|  defined = dom dict
----------------------------------------

end specification
```

The ZSL input form is discussed in detail in section 4.

The complete specification of *DataDictionary*, as well as several other sample specifications in all three input styles are included in the ZTC distribution.

3 The LaTeX Input — Basics

zed and oz are two LaTeX packages for typesetting Z specifications. Z specifications prepared using oz or zed can be type checked by ZTC, perhaps with some minor modifications. To use the LaTeX input form, some knowledge of LaTeX is necessary. If you are not familiar with LaTeX but want to learn about LaTeX, consult Leslie Lamport's *LaTeX: A Document Preparation System* [9]. Otherwise, skip this section, and use ZSL instead. You will have to get familiar with zed or oz before using ZTC. The zed package is described in Mike Spivey's *A Guide to the* zed *Style Option* [8], and the oz package is described in Paul King's *Printing Z and Object-Z LaTeX Documents*[7].

3.1 Choosing a package

ZTC now accepts specification written with either zed and oz packages. The two packages are mostly compatible but not completely. The oz package has better mnemonic names and includes all the $\mathcal{AMS}$ mathematical symbols, which will be handy when you use user-defined symbols in your specification. Furthermore, oz can be used to typeset Object-Z specifications as well as plain Z, although ZTC only accepts plain Z. Another difference of the two packages is due to the fact that LaTeX is currently undergoing a transition from LaTeX2.09 to LaTeX2ε. The zed package is not compatible with the current standard LaTeX2ε, while the oz package is distributed as a supported component of LaTeX2ε.

ZTC supports the following package-selection modes:

a) zed mode.
   Use the zed package. You can only use the commands defined in zed.

b) oz-zed compatible mode.
   Use the oz package. You can use all the commands defined in oz and zed. Incompatibilities are resolved in favor of zed.

c) oz native mode.
   Use the oz package. You can only use the commands defined in oz.

The incompatible commands in zed and oz are listed in Figure 1.

symbol	zed command	oz command
$\varnothing$	`\empty`	`\emptyset`
$\widehat{=}$	`\defs`	`\sdef`
==	`==`	`\defs`

Figure 1: Incompatible commands between zed and oz.

In the LaTeX preamble, you must

1. *indicate whether* `zed` *or* `oz` *package is used;*
2. *use the* `ztc` *package included in the distribution; (*`ztc` *must follow* `zed` *or* `oz`*.)*
3. *inform ZTC about your package selection decision using ZTC pragmas:*
   - `zed` *mode:*
     None. This is the default mode.
   - `oz-zed` *compatible mode:*
     ```
     \zedcompatible
     %% oz
     ```
   - `oz` *native mode:*
     ```
     %% oz-native
     ```

Here are same examples of LaTeX preamble.

1. Using `zed` with LaTeX2.09.

   ```
   \documentstyle[zed,ztc]{article}
   ```

2. Using `oz-zed` compatible mode with LaTeX2ε.

   ```
   \documentclass{article}
   \usepackage{oz,ztc}
   \zedcompatible
   %% oz
   ```

3. Using `oz` native mode with LaTeX2ε.

   ```
   \documentclass{article}
   \usepackage{oz,ztc}
   %% oz-native
   ```

Note that, the `oz` and `oz-native` pragmas must be in the preamble for them to take effect.

3.2 Lexical conventions

The lexical elements of LaTeX input form can be classified into the following categories:

a) *LaTeX commands*, which begin with a backslash (\), such as `\begin`, `\power_1`.

b) *Keywords*, such as `schema`, `zed`;

c) *Identifiers*, such as `DataDictionary`, `name?`;

d) *Integers*, such as `0`, `65535`;

e) *Symbols*, which consists of one or more non-alphanumeric characters, such as: `:`, `==`, `::=`.

The LaTeX input form is case sensitive.

> *The rules for forming identifiers in the LaTeX input form are the following:*
>
> - *An* identifier *consists of a* word *followed by a possibly empty* decoration.
> - *A* word *can be in one of the following forms:*
>   - *a letter followed by zero or more letters, digits, or underscores* (`\_`)*;*
>   - *a LaTeX command, i.e., a back slash* (`\`) *followed by letters.*
> - *A* decoration *is sequence of zero or more stroke characters,* `'`, `?`, `!`*, or subscriptions.*

Here are some examples of legal identifiers in LaTeX input form:

$ident1$	`ident1`
$ident_2$	`ident_2`
$ident'_0$	`ident'_0`
ε	`\varepsilon`
max_size	`max\_size`

The only primitive data type Z supports is the integer type. ZTC recognizes signed and unsigned decimal integers.

The mnemonic names of Z symbols are listed in Appendix A.

3.3 Structure of specifications

ZTC can directly type check LaTeX source files containing Z specifications.

> *Each specification must be enclosed by one of the following environments:*
>
> - `\begin{spec}` ... `\end{spec}`
> - `\begin{document}` ... `\end{document}`
>
> *Everything outside these environments are ignored by ZTC.*

The `spec` environment can occur multiple times and can be nested. This is useful when you want to put several specifications in a single document, or to divide a large specification into separate files and type check them separately. The `document` environment can occur only once in a LaTeX document, and must be the outer-most environment. The `document` environment is retained for backward compatibility with version 1.3.

3.4 Formal and informal text

A Z specification consists of formal and informal text. ZTC will type check the formal text and ignore the informal text.

3.4.1 Formal environments

Formal text must be enclosed in one of the following formal *environments:*

- `\begin{axdef}` ... `\end{axdef}`
- `\begin{gendef}` ... `\end{gendef}`
- `\begin{schema}` ... `\end{schema}`
- `\begin{syntax}` ... `\end{syntax}`
- `\begin{zed}` ... `\end{zed}`

Anything outside these formal environments are considered informal text and ignored.

Note!

Specifically, formulae enclosed in `$...$`, `\(...\)`, and `\[...\]` are considered informal text and ignored.

Usage of these formal environments are briefly illustrated below.

The `axdef` environment is used to define the *axiom boxes*.

```
\begin{axdef}
    MaxSize: \nat
\where
    MaxSize \leq 65535
\end{axdef}
```

$MaxSize : \mathbb{N}$

$MaxSize \leq 65535$

The `gendef` environment is used to define the *generic boxes*.

```
\begin{gendef}{X,Y}
    First: X \cross Y \fun X
\where
    \forall x: X; y: Y @ First(x,y) = x
\end{gendef}
```

$[X, Y]$

$First : X \times Y \rightarrow X$

$\forall x : X;\ y : Y \bullet First(x, y) = x$

The `schema` environment is used to define the *schema boxes*.

```
\begin{schema}{InsertOk}
    \Delta DataDictionary \\
    name? : NAME \\
    info? : INFO \\
    resp! : Response
\where
    name? \notin defined \\
    \# defined < MaxSize \\
    dict' = dict \cup \{
      name? \mapsto info? \} \\
    resp! = Success
\end{schema}
```

InsertOk

$$\Delta DataDictionary$$
$$name? : NAME$$
$$info? : INFO$$
$$resp! : Response$$

$$name? \notin defined$$
$$\# defined < MaxSize$$
$$dict' = dict \cup \{name? \mapsto info?\}$$
$$resp! = Success$$

The `syntax` environment is used to define *free types*.

A `syntax` environment contains a sequence of syntax rules separated by the `\also` command.

```
\begin{syntax}
   OP & ::= & plus | minus | times | divide
\also
   EXP & ::= & const \ldata \nat \rdata \\
       & | & binop \ldata OP \cross
             EXP \cross EXP \rdata
\end{syntax}
```

$$OP ::= plus \mid minus \mid times \mid divide$$
$$EXP ::= const\langle\!\langle \mathbb{N} \rangle\!\rangle \mid binop\langle\!\langle OP \times EXP \times EXP \rangle\!\rangle$$

The `zed` environment is used to define other paragraphs in Z, including *given sets*, *schema definitions*, *equivalence definitions*, and *predicates*. Short free type definitions can also be included in the `zed` environment. A `zed` environment may contain several paragraphs. The paragraphs in a `zed` environment must be separated by the `\also` command.

```
\begin{zed}
   [ADDR, PAGE]
\also
   DataDictInit \defs [ DataDictionary' | \\
\t3     defined' = \empty ]
\also
   DATABASE == ADDR \fun PAGE
\also
   \exists n: NAME @ birthday(n) \in December
\also
   REPORT ::= ok | unknown \ldata NAME \rdata
\end{zed}
```

$$[ADDR, PAGE]$$
$$DataDictInit \mathrel{\widehat=} [DataDictionary' \mid defined' = \varnothing]$$
$$DATABASE == ADDR \rightarrow PAGE$$
$$\exists n : NAME \bullet birthday(n) \in December$$
$$REPORT ::= ok \mid unknown\langle\!\langle NAME \rangle\!\rangle$$

The formal environments may not be nested.

Note! You may use either `syntax` and `zed` environments to define free types, but they not inter-changable. Use the `syntax` environment for the vertical format, and use the `zed` environment for the horizontal format.

3.4.2 Comments inside formal environments

Sometimes, you may want to put comments inside the formal environments. This can be accomplished by using `\comm` or `\remark` commands

Informal comments or remarks inside formal environments can be introduced as follows:

- `\comm{` *informal text* `};`
- `\remark{` *informal text* `};`

ZTC ignores the arguments of these two commands.

These two commands are synonymous.

3.4.3 Ignore formal environments

Sometimes, you may want ZTC to ignore some formal text without deleting them. Formal text can be commented out using `comment` or `nocheck` environments.

Anything enclosed in the following environments, including formal environments, will be ignored by ZTC.

- `\begin{comment}` ... `\end{comment}`
- `\begin{nocheck}` ... `\end{nocheck}`

Note! It is no longer necessary to comment out informal text outside formal environments.

3.5 White spaces

ZTC recognizes some commonly used LaTeX spacing commands and ignores them.

ZTC *ignores the following LaTeX commands inside formal environments:*

- *spacing commands:* `~`, `\,`, `\!`, `\:`, `\;`, `{}`
- *blank lines.*

3.6 Separators

Separator are used to separate between declarations or predicates in the axiom, generic, and schema boxes.

ZTC *treats the following commands as separators in* `axdef`, `gendef`, *and* `schema` *environments:*

- *semi-colon* (`;`)
- *the* `\also` *command, and*
- *the LaTeX linebreaking commands,* `\\` *and* `\linebreak`.

Omission of separators between declarations or predicates will cause syntax and/or typing errors. However, extra separators cause no harm.

The following two examples are equivalent semantically, but differ in their printout.

a) Linebreak as separators:

```
\begin{schema}{InsertOk}
   \Delta DataDictionary \\ name? : NAME \\
   info? : INFO \\ resp! : Response
\where
   name? \notin defined \\
   \# defined < MaxSize \\
   dict’ = dict \cup
     \{ name? \mapsto info? \} \also
   resp! = Success
\end{schema}
```

InsertOk

$\Delta DataDictionary$
$name? : NAME$
$info? : INFO$
$resp! : Response$

$name? \notin defined$
$\# defined < MaxSize$
$dict' = dict \cup \{name? \mapsto info?\}$

$resp! = Success$

b) Semi-colon as separators:

```
\begin{schema}{InsertOk}
   \Delta DataDictionary; name? : NAME \\
   info? : INFO; resp! : Response
\where
   name? \notin defined ;
   \# defined < MaxSize \\
   dict’ = dict \cup
     \{ name? \mapsto info? \} \\
   resp! = Success
\end{schema}
```

InsertOk

$\Delta DataDictionary;\ name? : NAME$
$info? : INFO;\ resp! : Response$

$name? \notin defined;\ \# defined < MaxSize$
$dict' = dict \cup \{name? \mapsto info?\}$
$resp! = Success$

Note! The separator rule above does not apply to the `syntax` and `zed` environments.

> *Paragraphs in a* `zed` *environment and free type definitions in a* `syntax` *environment must be separated by the* `\also` *command.*

3.7 Line continuing command

Sometimes, you may want to break a line without terminating the current declaration or predicate, e.g., when you have a long predicate that can not fit into a single line. You can accomplish this by using the *line continuing command*.

> *A line continuing command is a linebreaking command followed by a TAB command, which is one of the following:*
> `\t0`, `\t1`, `\t2`, `\t3`, `\t4`, `\t5`, `\t6`, `\t7`, `\t8`, `\t9`.
> *Line continuing commands are treated as white spaces by ZTC.*

When you print out specifications, a continuation command will cause a linebreak and an indentation of the continuing line. The amount of space indented is determined by the TAB command, with

`\t1` indents the least and `\t9` the most amount of space. The TAB command is not only necessary for ZTC to perform type-checking properly, but also desirable for enhancing the readability of specifications. The following example shows the proper use of the line continuing command.

```
\begin{gendef}{X,Y}
    First: X \cross Y \fun X
\where
    \forall x: X; y: Y @ \\
\t1    First(x,y) = x
\end{gendef}
```

$[X, Y]$
$First : X \times Y \to X$
$\forall x : X;\ y : Y \bullet$
$\quad First(x, y) = x$

When the indentation is not desired, you can use `\t0` as in the example below. It is equivalent to the previous one, however the continuing line is not indented and the printout is less readable.

```
\begin{gendef}{X,Y}
    First: X \cross Y \fun X
\where
    \forall x: X; y: Y @ \\
\t0    First(x,y) = x
\end{gendef}
```

$[X, Y]$
$First : X \times Y \to X$
$\forall x : X;\ y : Y \bullet$
$First(x, y) = x$

The continuation commands can also be used in the `zed` environment to break a long paragraph as in the following example.

```
\begin{zed}
Insert \defs InsertOk \lor InsertOverflow \\
\t1 \lor InsertAlreadyDefined
\end{zed}
```

$Insert \mathrel{\widehat=} InsertOk \lor InsertOverflow$
$\quad \lor InsertAlreadyDefined$

3.8 File inclusion

ZTC allows you to break a long specification into several input files and then include them into a master file.

You may use either of the following commands to include a file:

- `\input{` *filename* `}`
- `\include{` *filename* `}`

The complete filename must be specified in the file inclusion commands. The file inclusion commands can be nested. The maximum depth of inclusion is 16.

4 The ZSL Input — Basics

4.1 Lexical conventions

The lexical elements of ZSL can be classified into following categories:

- *Keywords*, such as `schema`, `where`;
- *Identifiers*, such as `DataDictionary`, `name?`;
- *Integers*, such as `0`, `65535`;
- *Bars*, which are used in the box style, such as `=========`, `|`;
- *Symbols*, which consists of one or more non-alphanumeric characters, such as: `:`, `==`, `::=`.

ZSL is case sensitive.

> *The rules for forming identifiers are the following:*
> - *An* identifier *consists of a* word *followed by a possible empty* decoration.
> - *A* word *must begin with a letter and followed by letters, digits, or underscores* (_).
> - *A* decoration *consists of one or more stroke characters,* `'`, `?`, `!`, *or* subscriptions.
> - *A* subscription *consists of an underscore followed by a digit.*

Here are some examples of legal identifiers in ZSL:

```
ident1  ident_2  ident'_0
```

The only primitive data type Z supports is the integer type. ZTC recognizes signed and unsigned decimal integers.

The bars are used in the box style to form axiom, generic, and schema boxes. There are three types of bars:

- *vertical bars*: `|`;
- *horizontal single bars*: `---------`;
- *horizontal double bars*: `=========`.

> *The length of a horizontal bar must be at least 3. The actual length of a horizontal bar is insignificant.*

ZSL defines ASCII equivalents for all the mathematical symbols used in Z. They are listed in Appendix A.

4.2 Structure of specifications

A Z specification consists of a sequence of paragraphs.

> *A ZSL input file consists of a sequence of paragraphs enclosed by one of the following specification environment*
>
> - `specification` ... `end specification`, *or*
> - `spec` ... `end spec`
>
> *The paragraphs must be separated one or more blank lines.*

You can have multiple specification environments in a ZSL file. Specification environments can also be nested. Everything outside specification environments are ignored.

A complete ZSL input is shown earlier in Section 2 (page 3.)

Paragraphs can be classified into box or non-box paragraphs. A *box paragraph* is either an axiom box, a generic box, or a schema box. All other paragraphs are called *non-box paragraphs*.

4.3 Informal text and comments

ZSL allows mixed informal text and formal specifications in ZSL input files. Unlike most programming languages, in which indentation and vertical alignment are insignificant, in ZSL, indentation and vertical alignment are used to distinguish formal text form informal text.

> ZTC *treats any line beginning with a TAB as formal text, and any line not beginning with a TAB as informal text.*
> ZTC *also treat anything following a percentage sign* (%) *up to the end of the line as informal text and ignores them.*

The following example shows a ZSL schema with comments:

```
        schema DataDictionary
          dict : NAME +-> INFO;
          defined : P NAME
        where
          defined = dom dict;
          # defined <= MaxSize
        end schema
defined is the set of all terms defined in the
data dictionary.
```

4.4 Box paragraphs

The plain text style and box style of ZSL are only different for box paragraphs, and they are identical for all other syntactical structures. For the box paragraphs, the text style uses keywords, such as `schema ... where ... end schema`, to define the syntactical structures, whereas the box style uses horizontal and vertical bars to define the syntactical structures.

An axiom box can be written in the following forms:

Plain text style:

```
global
    MaxSize : N
axiom
    MaxSize <= 65535
end axiom
```

Box style:

```
|  MaxSize : N
|----------
|  MaxSize <= 65535
```

A generic box can be written in the following forms:

Plain text style:

```
generic [X,Y]
    First: X & Y --> X
where
    forall x: X; y: Y @ First(x,y) = x
end generic
```

Box style:

```
=== [X,Y] ================================
|   First: X & Y --> X
|----------
|   forall x: X; y: Y @ First(x,y) = x
------------------------------------------
```

A schema box can be written in the following forms:

Plain text style:

```
schema InsertOk
    Delta DataDictionary;
    name? : NAME;
    info? : INFO;
    resp! : Response
where
    name? notin defined;
    # defined < MaxSize;
    dict' = dict || { name? -> info? };
    resp! = Success
end schema
```

Box style:

```
--- InsertOk ---------------------------
|  Delta DataDictionary;
|  name? : NAME;
|  info? : INFO;
|  resp! : Response
|----------
|  name? notin defined;
|  # defined < MaxSize;
|  dict' = dict || { name? -> info? };
|  resp! = Success
----------------------------------------
```

When using the plain text style, spaces following the leading TAB of each line are allowed and they are ignored. However, when using the box style, spaces following the leading TAB are not allowed.

When using the box style, each line must begin with a TAB, and the box must immediately follow the TAB. No space in between is allowed.

This ensures that all the boxes are aligned vertically. The length of horizontal bars must be at least that 3. Other than that the length of the horizontal bars is insignificant.

4.5 Non-box paragraphs and expressions

The text and box styles of ZSL are identical for non-box paragraphs and expressions The non-box paragraphs include given sets, schema definition, equivalence definition, predicates, and free types. Non-box paragraphs can be written as follows:

```
[ADDR, PAGE]

DataDictInit is [ DataDictionary’ |
                  defined’ = {} ]

DATABASE == ADDR --> PAGE

exists n: NAME @  birthday(n) in December

OP ::=  plus | minus | times | divide

EXP ::=  const << N >>
      |  binop << OP & EXP & EXP >>
```

Consult Appendix A for the syntactical structures of non-box paragraphs. ZSL defines ASCII equivalents for all the mathematical symbols used in Z. They are also listed in Appendix A.

4.6 Separators

Separator are used to separate between declarations or predicates in a sequence of declarations or predicates in the axiom, generic, and schema boxes. Omission of separators between declarations or predicates will cause syntax and/or typing errors.

A sequence of declarations and predicates must be separated by semi-colons (`;`).

The example on the left is incorrect, since a new line is not considered as a separator in ZSL. The correct input is shown on the right.

```
schema InsertOk
  Delta DataDictionary
  name? : NAME
  info? : INFO
  resp! : Response
where
  name? notin defined
  # defined < MaxSize
  dict' = dict || { name? -> info? }
  resp! = Success
end schema
```

```
schema InsertOk
  Delta DataDictionary;
  name? : NAME;
  info? : INFO;
  resp! : Response
where
  name? notin defined;
  # defined < MaxSize;
  dict' = dict || { name? -> info? };
  resp! = Success
end schema
```

4.7 File inclusion

ZSL allows you to break a long specification into several input files and then include them into a master file.

You may use either of the following commands to include a file:

a) `input` *filename*

b) `include` *filename*

The complete filename must be specified in the file inclusion commands. The file inclusion commands can be nested. The maximum depth of inclusion is 16.

5 Advanced Features

All the features described in this section are new in version 2.0. Most of them apply to both input forms.

5.1 ZTC pragmas

ZTC recognizes a number of *pragmas*to support non-standard extension to Z and allow you to exert fine control over the behavior of ZTC.

> *A* pragma *begins with double percentage signs* (%%) followed by a single space and the pragma name. Zero or more arguments separated by spaces may follow.

Anything following a pragma on the same line will be considered parameters of the pragma, not part of formal text.

We have already seen the `oz` and `oz-native` in section 3.1.

```
%% oz
%% oz-native
```

Neither of them has arguments.

5.2 User-defined symbols

ZTC allows user-defined symbols. You can define

- infix relational symbols,
- prefix relational symbols,
- infix generic symbols,
- prefix generic symbols, and
- infix function symbols.[2]

[2] Postfix function symbols can be defined using LaTeX macros discussed in the next section.

User-defined relational and generic symbols are defined as follows:

- *infix relational symbol:*
  `%% inrel` symbol
- *prefix relational symbols:*
  `%% prerel` symbol
- *infix generic symbols:*
  `%% ingen` symbol
- *prefix generic symbols:*
  `%% pregen` symbol

A symbol *is either a* word *or 1 to 4 character combinations of the following characters:*

```
\ @ / < > = & | - + * : . _ { } ( )
[ ]
```

~ (ZSL only)

Here are some examples of user-defined relational and generic symbols.

```
%% inrel  \prec
%% prerel \odd
%% pregen \smallpower
```

When define a infix function symbol, ZTC also allow you to specify its priority and associativity.

User-defined infix function symbols are defined as follows:

`%% infun`<*associativity*><*priority*> symbol

where

- <*associativity*> *is either* `l`*, for left associative, or* `r`*, for right associative.*
- <*priority*> *is a digit from* 0 *to* 7 *indicating the priority of the symbol.*

Priority 0 is the lowest and 7 the highest. The priority of the pre-defined infix function symbols are shown in Figure 2. All the pre-defined infix function symbols are left-associative.

Here are some examples of user-defined infix function symbols.

```
%% infunl4 \times
%% infunr0 \myop
```

Note!

The `inrel`, `prerel`, `ingen`, `pregen`, and `infun` pragmas only define the lexical categories of the user-defined symbols. You have to define the type and meaning of the symbols before you can

Priority 1	$\mapsto$
Priority 2	$..$
Priority 3	$+ \ - \ \cup \ \setminus \ \frown \ \uplus \ \uminus$
Priority 4	$* \ \mathrm{div} \ \ \mathrm{mod} \ \cap \ \upharpoonleft \ \upharpoonright \ \fcmp \ \circ \ \otimes$
Priority 5	$\oplus \ \sharp$
Priority 6	$\lhd \ \rhd \ \ndres \ \nrres$

Figure 2: The priority of the pre-defined infix function symbols

use them. You may also have to define their visual appearances using `\def` or `\newcommand`, if they are not already defined.

Figure 3 shows an example of user-defined symbols.

Any identifier *Rel* that denotes a relation can be converted to a infix relational symbol <u>*Rel*</u>.

> *The infix relational symbol corresponding to* `Rel` *is written as*
> `\inrel{Rel}`

Here is an example.

```
\begin{axdef}
  divides: \nat_1 \rel \nat_1
\where
  \forall x, y : \nat_1 @ x \inrel{divides} y
     \iff x \mod y = 0
\end{axdef}
```

$$divides : \mathbb{N}_1 \leftrightarrow \mathbb{N}_1$$
$$\forall x, y : \mathbb{N}_1 \bullet x \ \underline{divides} \ y \Leftrightarrow x \bmod y = 0$$

5.3 Using LaTeX macros

ZTC allows Z expressions to be written in LaTeX macro syntax, so that they can be typeset in anyway you like.

> *The following LaTeX macro*
> `\xyz`$\{exp_1\}\{exp_2\}\ldots\{exp_n\}$
> *is interpreted by ZTC as*
>
> `\xyz`$(exp_1, exp_2, \ldots, exp_n)$

This feature can be used to define so-called *outfix* or *surround-fix* symbols as shown below:

```
%% inrel  \prec
%% prerel \odd
%% pregen \smallpower
%% infunl4 \times
\def \odd {\mathsf{Odd}~}
\def \smallpower {\bbold S~}

\begin{axdef}
\_ \prec \_ : \num \rel \num \\
\odd \_ : \power \num \\
\_ \times \_ : \nat_1 \cross \nat_1
    \pfun \nat_1
\where
\forall x, y : \num @
  x \prec y \iff x + 1 < y \\
\forall x : \num @
  \odd x \iff x \mod 2 = 1 \\
\forall x, y : \nat_1 @
  x \times y = x * y
\end{axdef}

\begin{zed}
\smallpower X ==
  \{ S : \power X | \# S \leq 10 \}
\end{zed}
```

$$\_ \prec \_ : \mathbb{Z} \leftrightarrow \mathbb{Z}$$
$$\mathsf{Odd}\ \_ : \mathbb{P}\,\mathbb{Z}$$
$$\_ \times \_ : \mathbb{N}_1 \times \mathbb{N}_1 \pfun \mathbb{N}_1$$
$$\forall x, y : \mathbb{Z} \bullet x \prec y \Leftrightarrow x + 1 < y$$
$$\forall x : \mathbb{Z} \bullet \mathsf{Odd}\ x \Leftrightarrow x \bmod 2 = 1$$
$$\forall x, y : \mathbb{N}_1 \bullet x \times y = x * y$$

$$\mathbb{S}\ X == \{S : \mathbb{P}\,X \mid \#S \leq 10\}$$

Figure 3: User-defined symbols.

```
\def \abs#1 {~|#1|~}

\begin{axdef}
\abs{\_} : \num \fun \nat
\where
\forall x : \num @
\abs{x} = \zif x \geq 0 \zthen x \\
\t4    \zelse -x
\end{axdef}
```

$$|\ \_\ | : \mathbb{Z} \rightarrow \mathbb{N}$$
$$\forall x : \mathbb{Z} \bullet\ |\ x\ | = \textbf{if } x \geq 0 \textbf{ then } x \ \textbf{else } -x$$

This feature can also be used to define postfix function symbols and fractions, etc.

The `ignore` and `null-token` pragmas will make LaTeX commands disappear.

The effect of the following `ignore` *pragma*

```
%% ignore \xyz
```

is to make

`\xyz`$\{exp_1\}\{exp_2\}\ldots\{exp_n\}$

equivalent to

$(exp_1, exp_2, \ldots, exp_n)$

```
\def\ace {\mathsf{A}}
\def\king {\mathsf{K}}
...
\def\two {\mathsf{2}}
Suppose we ...
\begin{zed}
SUIT ::= \spadesuit | \heartsuit |
  \diamondsuit | \clubsuit
\also
RANK ::= \ace | \king | \queen | \jack |
  \ten | \nine | \eight | \seven | \\
\t2 \six | \five | \four | \three | \two
\also
Cards == SUIT \cross RANK
\end{zed}
Thus, we can say
\begin{zed}
(\spadesuit, \ace) \in Cards
\end{zed}
Or we can say
\def\card#1#2{#1#2}
%% ignore \card
\begin{zed}
\card{\spadesuit}{\ace} \in Cards
\end{zed}
Is this more fun?
```

Suppose we are specifying a deck of playing cards. Each card consists of a suit and a rank. It can be specified as follows:

$$SUIT ::= \spadesuit \mid \heartsuit \mid \diamondsuit \mid \clubsuit$$

$$RANK ::= \mathsf{A} \mid \mathsf{K} \mid \mathsf{Q} \mid \mathsf{J} \mid \mathsf{10} \mid \mathsf{9} \mid \mathsf{8} \mid \mathsf{7} \mid \mathsf{6} \mid \mathsf{5} \mid \mathsf{4} \mid \mathsf{3} \mid \mathsf{2}$$

$$Cards == SUIT \times RANK$$

Thus, we can say

$$(\spadesuit, \mathsf{A}) \in Cards$$

Or we can say

$$\spadesuit\mathsf{A} \in Cards$$

Is this more fun?

Figure 4: The `Ignore` pragma.

Figure 4 shows an example of using the `ignore` pragma.

> *The effect of the* `null-token` *pragma is to discard a parameterless LaTeX command.*

The `null-token` pragma is useful in situations such as you want to customize the layout of the schemas. The example in Figure 5 shows how to center the predicates in schemas.

5.4 Liberal mode

ZTC 2.0 introduces the *liberal mode*, which is a deviation from the Z notation defined in ZRM. However, I believe that it is quite reasonable.

> *In the liberal mode, declarations of variables can be omitted as long as their types can be deduced from the context up to and including the current paragraph.*

By default, ZTC is in the *strict mode* that follows the strict rules given in ZRM.

```
\def \bstack {\begin{array}{c}}
\def \estack {\end{array}}
%% null-token \bstack
%% null-token \estack

\begin{schema}{S}
  x, y, z : \nat
\where
  \bstack
  x \geq 0 \\
  y \geq x \geq z
  \estack
\end{schema}
```

S

$x, y, z : \mathbb{N}$

$$\begin{array}{c} x \geq 0 \\ y \geq x \geq z \end{array}$$

Figure 5: `Null-token` pragma.

You can switch between the strict and liberal modes using the following pragmas:

- `%% liberal`*: to enter the liberal mode.*
- `%% strict`*: to enter the strict mode.*

You can also use the `-L` switch from the command line to set the default mode to the liberal mode.

A trivial example of using the liberal mode is the following:

```
%% liberal

\begin{zed}
x = 2 \also
y = x + 1 \also
u = v \cup \{ 1 \} \also
s = \{ x, y \} \also
f ( x, y ) = s
\end{zed}
```

$x = 2$

$y = z + 1$

$u = v \cup \{1\}$

$s = \{x, y\}$

$f(x, y) = s$

This specification is illegal in strict mode but legal in liberal mode. The variables are used without declaration but their types can be easily deduced from the context.

The following is a more sensible use of liberal mode. This is an excerpt from Susan Stepney's *High Integrity Compilation*[10], in which she specified the semantics of compilers using Z. She wrote:

> The definitions of the semantics functions are quantified over all the variables appearing on the left-hand side of the equation. ...
>
> The continual occurrence of such quantifications tends to clutter the specification. So this is abbreviated, by omitting the declarations of all the arguments of the meaning functions, whose types can easily be deduced. (p. 23)

In fact, such abbreviations are not only reasonable, but also very common and widely accepted in literatures. This example also illustrates the LaTeX macro feature of ZTC. Using $[\![\ldots]\!]$ instead of $(\ldots)$ to enclose the arguments is a standard convention when writing denotation semantics. Using the strict syntax required by Z will be less straightforward and socially unacceptable. With the LaTeX macros, you can follow the accepted conversions in the printout, and still type-check your specifications.

So instead of writing the following semantic function in the strict mode:

$$\mathcal{D}_{\mathcal{EXPR}}[\![\_]\!] : EXPR \to State \pfun \mathbb{Z}$$

$$\begin{array}{l} \forall \chi : \mathbb{Z};\ \xi : NAME;\ \epsilon, \epsilon_1, \epsilon_2 : EXPR;\ \omega : OP;\ \sigma : State \bullet \\ \quad \mathcal{D}_{\mathcal{EXPR}}[\![number\ \chi]\!]\sigma = \chi \\ \quad \land \mathcal{D}_{\mathcal{EXPR}}[\![variable\ \xi]\!]\sigma = \sigma\xi \\ \quad \land \mathcal{D}_{\mathcal{EXPR}}[\![negate\ \epsilon]\!]\sigma = -(\mathcal{D}_{\mathcal{EXPR}}[\![\epsilon]\!]\sigma) \\ \quad \land \mathcal{D}_{\mathcal{EXPR}}[\![operation(\epsilon_1, \omega, \epsilon_2)]\!]\sigma = \\ \qquad \mathcal{D}_{\mathcal{OP}}[\![\omega]\!](\mathcal{D}_{\mathcal{EXPR}}[\![\epsilon_1]\!]\sigma, \mathcal{D}_{\mathcal{EXPR}}[\![\epsilon_2]\!]\sigma) \end{array}$$

we can write it in the liberal mode as follows:

```
%% liberal

\def\dop#1{\mathcal{D}_\mathcal{OP}%
  \lbag #1 \rbag}
\def\dexpr#1{\mathcal{D}_\mathcal{EXPR}%
  \lbag #1 \rbag}
\begin{axdef}
  \dexpr{\_} : EXPR \fun State \pfun \num
\where
  \dexpr{number~\chi} \sigma = \chi \\
  \dexpr{variable~\xi} \sigma = \sigma \xi \\
  \dexpr{negate~\epsilon} \sigma =
    - (\dexpr{\epsilon} \sigma) \\
  \dexpr{operation(\epsilon_1, \omega,
    \epsilon_2)}
  \sigma = \\
\t1  \dop{\omega}(\dexpr{\epsilon_1} \sigma,
  \dexpr{\epsilon_2} \sigma)
\end{axdef}
```

$$\mathcal{D}_{\mathcal{EXPR}}[\![\_]\!] : EXPR \to State \pfun \mathbb{Z}$$

$$\begin{array}{l} \mathcal{D}_{\mathcal{EXPR}}[\![number\ \chi]\!]\sigma = \chi \\ \mathcal{D}_{\mathcal{EXPR}}[\![variable\ \xi]\!]\sigma = \sigma\xi \\ \mathcal{D}_{\mathcal{EXPR}}[\![negate\ \epsilon]\!]\sigma = -(\mathcal{D}_{\mathcal{EXPR}}[\![\epsilon]\!]\sigma) \\ \mathcal{D}_{\mathcal{EXPR}}[\![operation(\epsilon_1, \omega, \epsilon_2)]\!]\sigma = \\ \quad \mathcal{D}_{\mathcal{OP}}[\![\omega]\!](\mathcal{D}_{\mathcal{EXPR}}[\![\epsilon_1]\!]\sigma, \mathcal{D}_{\mathcal{EXPR}}[\![\epsilon_2]\!]\sigma) \end{array}$$

A more complete version of this example is included in the ZTC distribution.

5.5 Forward declaration of types

ZTC allows you to forward declare a type as a given set and then redefine it later.

A warning message will be issued for each redefinition.

5.6 Obtain type information

ZTC allows you to obtain type information of any expression.

The `typeof` *macro will print out the type of the its argument.*
`\typeof{` *exp* `}`

This feature is enabled by the `-T` switch and only available in the LaTeX input.

5.7 Verbosity control

You can control the verbosity of ZTC using the `verbose` pragma or the `V` switch from the command line.

The `verbose` *pragma sets the verbosity value from 0 to 9.*
`%% verbose[0-9]`

Verbosity value 0 is the least verbose and 9 is the most verbose. The default verbosity is 5.

6 Running ZTC

ZTC supports a number of command line options.

`ztc` [`-I` [`tl`]] *infile* [*options*]

On extended MS-DOS, use `ztc32` instead of `ztc`.

The input file name is required. It is no longer necessary to specify the input form when the full file name is given. ZTC will determine the input form automatically. You can still specify the input form explicitly with the `-I` switch, then the extension of the input file name may be omitted:

`-It` LaTeX form, the default extension is `zed`.

`-Il` ZSL form, the default extension is `zsl`.

The options of ZTC are:

`-F` Enable the flying-erase mode
This switch instructs ZTC to erase the paragraphs from the memory as soon as they have been type checked. It is particularly useful when you are running the standard DOS version. It increases the capacity of ZTC.

`-L` Set the default mode to the liberal mode.

`-M[0-9]` Select mathematical toolkit library.
`-M`*n* instructs ZTC to load `math`*n*`.zed` for LaTeX input and `math`*n*`.zbx` for ZSL input. The default mathematical toolkit library is `math1.zed` and `math1.zbx`.

Library `math0` contains the basic mathematical toolkit defined in ZRM. Library `math1` contains additional declarations of float, boolean, char, and string, see Appendix B.

`-M`*libfile* Select mathematical toolkit library.
This switch instructs ZTC to load *libfile*`.zed` for LaTeX input and *libfile*`.zbx` for ZSL input.

`-M-` Disable mathematical toolkit library.
This switch instructs ZTC not to load any mathematical toolkit library.

`-O[tlb]` *outfile* Translate the input file into a given output form and save the result to outfile. The extension of the output filename may be omitted

`-Ot` translate the input file to the LaTeX form, the default extension is `zed`.

`-Ol` translate the input file to the ZSL plain text form, the default extension is `zsl`.

`-Ob` translate the input file to the ZSL box form, the default extension is `zbx`.

When the translation is performed, only the formal text is translated, and all the informal text is deleted.

`-S` Suppress type-checking.

`-T` Generate a type report:
This switch generates a type report that contains the type information of all the names in the specification. The type report will be written in a file whose name is the same as the input file name with extension `.typ`.

`-V[0-9]` Set verbosity
The default verbosity is 5. 0 is the least verbose and 9 the most.

The single-letter options, `F, L, S,` and `T`, can be grouped togather, such as `-LT`.

Every time ZTC is invoked, a log file will be written. It contains all the messages sent to the standard output. The name of the log file is the same as the input file name with the extension `.log`.

Common usages:

1. `ZTC spec.zed`
   Type-checking LaTeX input file `spec.zed`.
2. `ZTC spec.zsl -T`
   Type-checking ZSL input file `spec.zsl`, and requesting a type report.
3. `ZTC spec.zed -Ob spec`
   Translating LaTeX input file `spec.zed` to ZSL box style. The output file will be named `spec.zbx`, and type-checking is performed on the input file.
4. `ZTC spec.zsl -Ot spec -S`
   Translating ZSL input file `spec.zsl` to LaTeX style. The output file will be named `spec.zed`, and type-checking is suppressed.

7 Installation

ZTC runs on the following platforms:

- Microsoft Windows 9x, NT, 2000, and XP, and
- Linux

7.1 The distribution package

The ZTC 2.1 distribution package contains the following files:

Documentation (all platforms)
`README`, a brief overview
`guide.ps`, this guide in PostScript
`ztc.1`, a man page

Executable files (one of the following sets)
`ztc`, for Linux
`ZTC.EXE`, for Windows/MS-DOS

Library files (all platforms)
`math0.zed`, for LaTeX input
`math0.zbx`, for ZSL input
`mathoz.zed`, for LaTeX input using `oz` package

LaTeX style file (all platforms)
`ztc.sty`, for LaTeX input

Sample Z specification files (all platforms)
`datadict.zed`, `datadict.zsl`, `datadict.zbx`

A simple *data dictionary* in three different input styles: LaTeX ZSL plain text style, and ZSL box style.

`liberal.zed`

A more complete version of the example on page 24 illustrating the liberal mode and LaTeX macro syntax for Z expressions.

`bridge.zed`

Some basic rules of bidding in contract bridge.

Registration form (all platforms)
`register.txt`, please send this in via e-mail.

7.2 Installing Windows/MS-DOS version

Step 1. Create a new directory on your hard drive for ZTC, say `C:\ZTC`.

Step 2. Copy the executable file `ZTC.EXE` to `\ZTC`.

Step 3. Copy the data files `MATH0.ZED`, `MATH0.ZBX`, and `MATHOZ.ZED` to `\ZTC`. (Alternatively, you can put them in the same directory where your specifications resides. Then you don't need to set `ZLIBPATH`.)

Step 4. If you use the LaTeX input form, copy the LaTeX style file `ZTC.STY` to the TeX input directory. If you use EmTeX it will be `C:\EMTEX\TEXINPUT`. You need to have `zed` or `oz` style package installed as well. (Alternatively, you can put it in the same directory where your specifications reside.)

Step 5. Update `AUTOEXEC.BAT` file by appending the following lines:

```
SET PATH=%PATH%;C:\ZTC;
SET ZLIBPATH=C:\ZTC
```

7.3 Installing Linux version

Assume that the executable directory is `/usr/local/bin`, and the data directory is usually `/usr/local/lib`.

Step 1. Copy the executable file `ztc` to `/usr/local/bin`.

Step 2. Copy the data file `math0.zed`, `math0.zbx`, and `mathoz.zed` to `/usr/local/lib`. (Alternatively, you can put them in the same directory where your specifications resides. Then you don't need to set `ZLIBPATH`.)

Step 3. If you use the LaTeX input form, copy the LaTeX style file `ztc.sty` to the TeX input directory, probably `/usr/local/lib/texmf/tex/latex2e` depending on your TeX installation and LaTeX version. You need to have `zed` or `oz` style package installed as well. (Alternatively, you can put it in the same directory where your specifications reside.)

Step 4. Make sure `/usr/local/bin` is in your search path.

Step 5. Set the environment variable `ZLIBPATH` to `/usr/local/lib`.

In `csh` and `tcsh` do:

```
setenv ZLIBPATH /usr/local/lib
```

In `bash`, `ksh`, and `sh` do:

```
ZLIBPATH=/usr/local/lib; export ZLIBPATH
```

You may want to put this in your shell initialization script.

8 Registration and Bug Reports

Please fill out the *Registration Form* included in the distribution package, and email it to

`jia@cs.depaul.edu`

You will receive information regarding new releases of ZTC and other tools for Z.

Comments on ZSL and ZTC are greatly appreciated. Send your comments and bug reports to the same address above. When filing a bug report, please include the following information:

a) hardware platform and operating system;

b) version of ZTC;

c) input file;

d) command line used to invoke ZTC.

References

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

[2] I. Hayes (ed.), *Specification Case Studies*, Prentice Hall International, 2nd edition, 1993.

[3] I. Houston, and S. King, "CICS Project: Experiences and Results From the Use of Z in IBM", *Proc. VDM'91 – Formal Software Development Methods*, LNCS No. 552, pp. 588-596, 1991.

[4] J.M. Spivey, "An Introduction to Z and Formal Specification," *Software Engineering Journal*, Vol. 4, No. 1, January 1989, pp. 40-50.

[5] J.B. Wordsworth, *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering*, Addison-Wesley, 1992.

[6] A. Diller, *Z: An Introduction to Formal Methods*, 2nd edition, John Wiley & Sons, 1994.

[7] P. King, *Printing Z and Object-Z LaTeX Documents*, 1990. Included in LaTeX2εdistribution. Available at CTAN cites.

[8] J.M. Spivey, *A guide to the* `zed` *style option*, 1990. Available via anonymous FTP at `ftp.comlab.ox.ac.uk`.

[9] L. Lamport, *LaTeX: A Document Preparation System*, 2nd edition, Addison-Wesley, 1994.

[10] S. Stepney, *High Integrity Compilation*, Prentice Hall, 1993.

A LaTeX and ZSL Input Notations

A.1 Paragraphs

A.1.1 Axiom Box

$$\begin{array}{|l} D_1;\ \ldots;\ D_m \\ \hline P_1;\ \ldots;\ P_n \end{array}$$

LaTeX input:

```
\begin{axdef}
   D_1; ... ; D_m
\where
   P_1; ... ; P_n
\end{axdef}
```

ZSL input – text style:

```
global
   D1; ... ; Dm
axiom
   P1; ... ; Pn
end axiom
```

ZSL input – box style:

```
|  D1; ... ; Dm
|-------
|  P1; ... ; Pn
```

$$\begin{array}{|l} D_1;\ \ldots;\ D_m \end{array}$$

LaTeX input:

```
\begin{axdef}
   D_1; ... ; D_m
\end{axdef}
```

ZSL input – text style:

```
global
   D1; ... ; Dm
end global
```

ZSL input – box style:

```
|  D1; ... ; Dm
```

$P_1; \ldots; P_n$

LaTeX input:

```
\begin{zed}
   P_1; ... ; P_n
\end{zed}
```

ZSL input

```
axiom
   P1; ... ; Pn
end axiom
```

A.1.2 Schema Box

S

$D_1; \ldots; D_m$

$P_1; \ldots; P_n$

LaTeX input:

```
\begin{schema}{S}
   D_1; ... ; D_m
\where
   P_1; ... ; P_n
\end{schema}
```

ZSL input – text style:

```
schema S
   D1; ... ; Dm
where
   P1; ... ; Pn
end schema
```

ZSL input – box style:

```
--- S --------------------
|  D1; ... ; Dm
|-------
|  P1; ... ; Pn
--------------------------
```

A.1.3 Generic Schema Box

$S[X_1, \ldots, X_k]$

$D_1; \ldots; D_m$

$P_1; \ldots; P_n$

LaTeX input:

```
\begin{schema}{S[X_1,...X_k]}
   D_1; ... ; D_m
\where
   P_1; ... ; P_n
\end{schema}
```

ZSL input – text style:

```
schema S [X1, ... , Xk]
   D1; ... ; Dm
where
   P1; ... ; Pn
end schema
```

ZSL input – box style:

```
--- S [X1, ... , Xk] -----
|  D1; ... ; Dm
|-------
|  P1; ... ; Pn
--------------------------
```

A.1.4 Generic Box

$[X_1, \ldots, X_k]$

$D_1; \ \ldots; \ D_m$

$P_1; \ \ldots; \ P_n$

LaTeX input `zed`:

```
\begin{gendef}[X_1,...,X_k]
   D_1; ... ; D_m
\where
   P_1; ... ; P_n
\end{gendef}
```

LaTeX input `oz`:

```
\begin{gendef}{X_1,...,X_k}
   D_1; ... ; D_m
\where
   P_1; ... ; P_n
\end{gendef}
```

ZSL input – text style:

```
generic [X1, ... , Xk]
   D1; ... ; Dm
where
   P1; ... ; Pn
end generic
```

ZSL input – box style:

```
=== [X1, ... , Xk] ==========
|  D1; ... ; Dm
|-------
|  P1; ... ; Pn
-----------------------------
```

A.1.5 Schema Definition

$S \mathrel{\widehat{=}} [D \mid P]$	LaTeX `zed`
	`S \defs [ D | P ]`
		ZSL
		`S =^= [ D | P ]`
		`S is [ D | P ]`

	LaTeX `oz` `S \sdef [ D | P ]`	

A.1.6 Given Set

$[T_1, \ldots, T_n]$	LaTeX `[T_1, ..., T_n]`	ZSL `[T1, ..., Tn]`

A.1.7 Equivalence Definition

$id == Exp$	LaTeX `zed` `id == Exp`	ZSL `id == Exp`
	LaTeX `oz` `id \defs Exp`	

A.1.8 Free Type Definition

$$\begin{aligned} T ::= {} & c_1 \mid \ldots \mid c_m \\ \mid {} & d_1 \langle\!\langle E_1[T] \rangle\!\rangle \\ \mid {} & \ldots \\ \mid {} & d_n \langle\!\langle E_n[T] \rangle\!\rangle \end{aligned}$$

LaTeX input `zed`:

```
\begin{syntax}
T & ::= & c_1 | ... | c_m \\
  & | & d_1 \ldata E_1[T] \rdata \\
  & | & ... \\
  & | & d_n \ldata E_n[T] \rdata
\end{syntax}
```

LaTeX input oz:

```
\begin{syntax}
T & \ddef & c_1 | ... | c_m \\
  & | & d_1 \lang E_1[T] \rang \\
  & | & ... \\
  & | & d_n \lang E_n[T] \rang
\end{syntax}
```

ZSL input:

```
T ::= c1 | ... | cm
    | d1 << E1[T] >>
    | ...
    | dn << En[T] >>
```

A.1.9 Schema Expressions

		LaTeX	ZSL
$\forall D \mid P \bullet S$		`\forall D	P @ S`	`forall D	P @ S`
	oz only ▶	`\all D	P \dot S`	
$\exists D \mid P \bullet S$		`\exists D	P @ S`	`exists D	P @ S`
	oz only ▶	`\exi D	P \dot S`	
$\exists_1 D \mid P \bullet S$		`\exists_1 D	P @ S`	`exists1 D	P @ S`
	oz only ▶	`\exione D	P \dot S`	
$[D \mid P]$		`[D	P]`	`[D	P]`
ΔS		`\Delta S`	`Delta S`
ΞS		`\Xi S`	`Xi S`
$S[T_1, ..., T_n]$		`S[T_1,...,T_n]`	`S[T1,...,Tn]`
$S[x_1/y_1, ..., x_n/y_n]$		`S[x_1/y_1,...,x_n/y_n]`	`S[x1/y1,...,xn/yn]`
$\mathrm{pre}\, S$		`\pre S`	`pre S`
$\neg S$		`\lnot S`	`not S`
$S_1 \land S_2$		`S_1 \land S_2`	`S1 and S2`
			`S1 /\ S2`
$S_1 \lor S_2$		`S_1 \lor S_2`	`S1 or S2`
			`S1 \/ S2`
$S_1 \Rightarrow S_2$		`S_1 \implies S_2`	`S1 implies S2`
	oz only ▶	`S_1 \imp S_2`	`S1 => S2`
$S_1 \Leftrightarrow S_2$		`S_1 \iff S_2`	`S1 iff S2`
			`S1 <=> S2`
$S_1 \upharpoonright S_2$		`S_1 \project S_2`	`S1 project S2`
			`S1 |\ S2`
$S \setminus (v_1, \ldots, v_n)$		`S \hide (v_1,...,v_n)`	`S hide (v1,...,vn)`
	oz only ▶	`S \zhide (v_1,...,v_n)`	`S \\ (v1,...,vn)`
$S_1 \fatsemi S_2$		`S_1 \semi S_2`	`S1 semi S2`
	oz only ▶	`S_1 \zcmp S_2`	`S1 // S2`
$S_1 \gg S_2$		`S_1 \pipe S_2`	`S1 pipe S2`
	oz only ▶	`S_1 \zpipe S_2`	

A.1.10 Predicates

		LaTeX	ZSL
$\forall D \mid P \bullet Q$		`\forall D	P @ Q`	`forall D	P @ Q`
	oz only ▶	`\all D	P \dot S`	
$\exists D \mid P \bullet Q$		`\exists D	P @ Q`	`exists D	P @ Q`
	oz only ▶	`\exi D	P \dot S`	
$\exists_1 D \mid P \bullet Q$		`\exists_1 D	P @ Q`	`exists1 D	P @ Q`
	oz only ▶	`\exione D	P \dot S`	
let $v == e \bullet P$		`\zlet v==e @ P`	`let v==e @ P`
	oz only ▶	`\zlet v==e \dot P`	
$p \land q$		`p \land q`	`p and q`
			`p /\ q`
$p \lor q$		`p \lor q`	`p or q`
			`p \/ q`
$p \Rightarrow q$		`p \implies q`	`p implies q`
	oz only ▶	`p \imp q`	`p => q`
$p \Leftrightarrow q$		`p \iff q`	`p iff q`
			`p <=> q`
$\neg p$		`\lnot p`	`not p`
true		`true`	`true`
			`TRUE`
false		`false`	`false`
			`FALSE`

A.2 Expressions

A.2.1 Lambda Expression

		LaTeX	ZSL
$\lambda D \mid P \bullet E$		`\lambda D	P @ E`	`lambda D	P @ E`

A.2.2 Definite Description

		LaTeX	ZSL
$\mu D \mid P \bullet E$		`\mu D	P @ E`	`mu D	P @ E`
			`unique D	P @ E`
	oz only ▶	`\mu D	P \dot E`	

A.2.3 Conditional expression

		LaTeX	ZSL
if P **then** E_1 **else** E_2		`\zif P \zthen E_1` `\zelse E_2`	`if P then E1 else E2`

A.2.4 Local definition

		LaTeX	ZSL
let $v == e \bullet E$		`\zlet v==e @ E`	`let v==e @ E`
	oz only ▶	`\zlet v==e \dot E`	

A.2.5 Sets

		LaTeX	ZSL
$\{x_1, \ldots, x_n\}$		`\{ x_1, ... , x_n \}`	`{ x1, ..., xn }`
$\{D \mid P \bullet E\}$		`\{ D | P @ E \}`	`{ D | P @ E }`
	oz only ▶	`\{ D | P \dot E \}`	
$S_1 \times S_2$		`S_1 \cross S_2`	`S1 & S2`
$S_1 = S_2$		`S_1 = S_2`	`S1 = S2`
$S_1 \neq S_2$		`S_1 \neq S_2`	`S1 /= S2`
$x \in S$		`x \in S`	`x in S`
	oz only ▶	`x \mem S`	
$x \notin S$		`x \notin S`	`x notin S`
	oz only ▶	`x \nem S`	
$\varnothing$		`\empty`	`{}`
$S_1 \subset S_2$		`S_1 \subset S_2`	`S1 subset S2`
	oz only ▶	`S_1 \psubs S_2`	
$S_1 \subseteq S_2$		`S_1 \subseteq S_2`	`S1 subseteq S2`
	oz only ▶	`S_1 \subs S_2`	
$\mathbb{P} S$		`\power S`	`P S`
	oz only ▶	`\pset S`	
$\mathbb{P}_1 S$		`\power_1 S`	`P1 S`
	oz only ▶	`\psetone S`	
$\mathbb{F} S$		`\finset S`	`F S`
	oz only ▶	`\fset S`	
$\mathbb{F}_1 S$		`\finset_1 S`	`F1 S`
	oz only ▶	`\fsetone S`	
$S_1 \cup S_2$		`S_1 \cup S_2`	`S1 setunion S2`
	oz only ▶	`S_1 \uni S_2`	`S1 || S2`
$S_1 \cap S_2$		`S_1 \cap S_2`	`S1 setint S2`
	oz only ▶	`S_1 \int S_2`	`S1 && S2`
$S_1 \setminus S_2$		`S_1 \setminus S_2`	`S1 setminus S2` `S1 \ S2`
$\bigcup SS$		`\bigcup SS`	`Union SS`

$\bigcap SS$		\bigcap SS	Intersection SS

A.2.6 Ordered Pairs

		LaTeX	ZSL
$x \mapsto y$		x \mapsto y	x mapsto y
	oz only ▶	x \map y	x -> y
$\mathit{first}\ P$		first P	first P
$\mathit{second}\ P$		second P	second P

A.2.7 Relations

		LaTeX	ZSL
$A \leftrightarrow B$		A \rel B	A <-> B
			A rel B
$x \underline{R} y$		x \inrel{R} y	x _R_ y
$\mathrm{dom}\, R$		\dom R	dom R
$\mathrm{ran}\, R$		\ran R	ran R
$\mathrm{id}\, S$		\id S	id S
$R_1 \fatsemi R_2$		R_1 \comp R_2	R1 comp R2
	oz only ▶	R_1 \fcmp R_2	R1 :> R2
$R_1 \circ R_2$		R_1 \circ R_2	R1 backcomp R2
	oz only ▶	R_1 \cmp R_2	R1 <: R2
$R_1 \lhd R_2$		R_1 \dres R_2	R1 dres R2
			R1 <| R2
$R_1 \ndres R_2$		R_1 \ndres R_2	R1 dsub R2
	oz only ▶	R_1 \dsub R_2	R1 <+ R2
$R_1 \rhd R_2$		R_1 \rres R_2	R1 rres R2
			R1 |> R2
$R_1 \nrres R_2$		R_1 \nrres R_2	R1 rsub R2
	oz only ▶	R_1 \rsub R_2	R1 +> R2
$R_1 \oplus R_2$		R_1 \oplus R_2	R1 oplus R2
	oz only ▶	R_1 \fovr R_2	R1 += R2
$R(\!\| S \|\!)$		R \limg S \rimg	R (| S |)
$R^{\sim}$		R \inv	R~
			R inversion
R^*		R \star	R^*
	oz only ▶	R \rtcl	R rtclosure
R^+		R \plus	R^+
	oz only ▶	R \tcl	R tclosure
R^k		R \bsup k \esup	R^(k)

A.2.8 Functions

		LaTeX	ZSL
$A ⇸ B$		`A \pfun B`	`A +-> B`
			`A pfun B`
$A \rightarrow B$		`A \fun B`	`A --> B`
	oz only ▶	`A \tfun B`	`A fun B`
$A ⤔ B$		`A \pinj B`	`A >+> B`
			`A pinj B`
$A \rightarrowtail B$		`A \inj B`	`A >-> B`
	oz only ▶	`A \tinj B`	`A inj B`
$A ⤀ B$		`A \psurj B`	`A +>> B`
	oz only ▶	`A \psur B`	`A psurj B`
$A \twoheadrightarrow B$		`A \surj B`	`A ->> B`
	oz only ▶	`A \tsur B`	`A surj B`
$A ⤖ B$		`A \bij B`	`A >->> B`
			`A bij B`
$A ⇻ B$		`A \ffun B`	`A ++> B`
			`A ffun B`
$A ⤕ B$		`A \finj B`	`A >++> B`
			`A finj B`

A.2.9 Numbers

		LaTeX	ZSL
$\mathbb{N}$		`\nat`	`N`
			`Nat`
$\mathbb{N}_1$		`\nat_1`	`N1`
	oz only ▶	`\natone`	`Nat1`
$\mathbb{Z}$		`\num`	`Z`
	oz only ▶	`\integer`	`Int`
$n \mathinner{\ldotp\ldotp} m$		`n \upto m`	`n upto m`
			`n .. m`
$x + y$		`x + y`	`x + y`
$x - y$		`x - y`	`x - y`
$x * y$		`x * y`	`x * y`
$x = y$		`x = y`	`x = y`
$x \neq y$		`x \neq y`	`x /= y`
$x \mathrm{div} y$		`x \div y`	`x div y`
$x \mathrm{mod} y$		`x \mod y`	`x mod y`
$x < y$		`x < y`	`x < y`
$x \leq y$		`x \leq y`	`x <= y`
$x > y$		`x > y`	`x > y`
$x \geq y$		`x \geq y`	`x >= y`
$succ\ x$		`succ x`	`succ x`

$\#S$		`\# S`	`# S`
min S		`min~S`	`min S`
max S		`max~S`	`max S`

A.2.10 Sequences

		LaTeX	ZSL
$\mathrm{seq}\,X$		`\seq X`	`seq X`
$\mathrm{seq}_1\,X$		`\seq_1 X`	`seq1 X`
	oz only ▶	`\seqone X`	
$\mathrm{iseq} X$		`\iseq X`	`iseq X`
$\langle s_1, \ldots, s_n \rangle$		`\langle s_1,...,s_n \rangle`	`<< s1,...,sn >>`
	oz only ▶	`\lseq s_1,...,s_n \rseq`	
$s \frown t$		`s \cat t`	`s concat t` `s ^ t`
head s		`head~s`	`head s`
last s		`last~s`	`last s`
tail s		`tail~s`	`tail s`
front s		`front~s`	`front s`
rev s		`rev~s`	`rev s`
$s \upharpoonright X$		`s \filter X`	`s filter X`
	oz only ▶	`s \sres X`	`s |- X`
$X \upharpoonleft s$		`X \extract s`	`X extract s`
	oz only ▶	`X \ires s`	`X -| s`
$\frown/\ ss$		`\dcat ss`	`^/ ss`
disjoint *ss*		`\disjoint ss`	`disjoint ss`
ss partition *S*		`ss \partition S`	`ss partition S`
s_1 in s_2		`s_1 \subseq s_2`	`s1 subseq s2`
	oz only ▶	`s_1 \inseq s_2`	
s_1 prefix s_2		`s_1 \prefix s_2`	`s1 prefix s2`
s_1 suffix s_2		`s_1 \suffix s_2`	`s1 suffix s2`
squash s		`squash~s`	`squash s`

A.2.11 Bags

	LaTeX	ZSL
$\mathrm{bag}\,X$	`\bag X`	`bag X`
$[\![a_1, \ldots, a_n]\!]$	`\lbag a_1,...,a_n \rbag`	`[[ a1,...,an ]]`
$x \sqsubset\!\!\in B$	`x \inbag B`	`x inbag B`
count B	`count B`	`count B`
$B_1 \sqsubseteq B_2$	`B_1 \subbag B_2`	`B1 subbag B2`
$B_1 \uplus B_2$	`B_1 \bagdiff B_2`	`B1 bagdiff B2`

		`B1 -- B2`
$n \otimes B$	`n \bagscale B`	`n bagscale B`
$B \sharp x$	`B \bagcount x`	`B bagcount x`
$B_1 \uplus B_2$	`B_1 \uplus B_2`	`B1 bagunion B2`
oz only ▶	`B_1 \buni B_2`	`B1 ++ B2`
items s	`items s`	`items s`

A.2.12 Binding

	LaTeX	ZSL
θS	`\theta S`	`theta S`

A.2.13 Selection

	LaTeX	ZSL
$S.x$	`S.x`	`S.x`

A.2.14 Operators

	LaTeX	ZSL
PreSym _	`PreSym \_`	`PreSym _`
_ *InSym* _	`\_ InSym \_`	`_ InSym _`
_ *PostSym*	`\_ PostSym`	`_ PostSym`
_ ⦇ _ ⦈	`\_ \limg \_ \rimg`	`_ (| _ |)`

Index