ProofPower

DESCRIPTION

PPTex-2.9.1w2.rda.110727

Information on the current status of ProofPower is available on the World-Wide Web, at URL:

`http://www.lemma-one.demon.co.uk/ProofPower/index.html`

This document is published by:

Lemma 1 Ltd.
2nd Floor
31A Chain Street
Reading
Berkshire
UK
RG1 2HX
e-mail: `pp@lemma-one.com`

PPTex-2.9.1w2.rda.110727 13:54:16 27/07/2011

CONTENTS

List of Tables

ABOUT THIS PUBLICATION

0.1 Purpose

This document will eventually provide a full narrative description of ProofPower to complement the catalogue of detail provided by the ProofPower *Reference Manual* [11]. It is at present incomplete, containing only those parts of its intended content which are least well covered by other ProofPower user documentation. Specifically, this version of the manual describes the concrete syntax of the specification languages supported by ProofPower.

0.2 Readership

Users of ProofPower.

0.3 Related Publications

A bibliography is given at the end of this document.

Publications relating specifically to ProofPower include:

1. ProofPower Tutorial *[7]*, tutorial covering the basic ProofPower system.
2. ProofPower Z Tutorial *[9]*, tutorial covering ProofPower Z support option.
3. ProofPower HOL Tutorial Notes *[10]*;
4. ProofPower Reference Manual *[11]*;
5. ProofPower Installation and Operation *[8]*;
6. ProofPower Document Preparation *[6]*.

0.4 Assumptions

Though it is not strictly prerequisite, the user may find this volume easier to read after attending a course on ProofPower, or after consulting the ProofPower *Tutorial* [7].

0.5 Acknowledgements

ICL gratefully acknowledges its debt to the many researchers (both academic and industrial) who have provided intellectual capital on which ICL has drawn in the development of ProofPower.

We are particularly indebted to Mike Gordon of The University of Cambridge, for his leading role in some of the research on which the development of ProofPower has built, and for his positive attitude towards industrial exploitation of his work.

The ProofPower system is a proof tool for Higher Order Logic which builds upon ideas arising from research carried out at the Universities of Cambridge and Edinburgh, and elsewhere.

In particular the logic supported by the system is (at an abstract level) identical to that implemented in the Cambridge HOL system [1], and the paradigm adopted for implementation of proof support for the language follows that adopted by Cambridge HOL, originating with the LCF system developed at Edinburgh [2]. The functional language "standard ML" used both for the implementation and as a interactive metalanguage for proof development, originates in work at Edinburgh, and has been developed to its present state by an international group of academic and industrial researchers. The implementation of Standard ML on which ProofPower is based was itself originally implemented by David Matthews at the University of Cambridge, and is now commercially marketed by Abstract Hardware Limited.

The ProofPower system also supports specification and proof in the Z language, developed at the University of Oxford. We are therefore also indebted to the research at Oxford (and elsewhere) which has contributed to the development of the Z language.

Standard ML of New Jersey is an implementation of the Standard ML language developed by Lucent. It is copyright © 1989–1998 by Lucent Technologies. The following licence terms apply to Standard ML of New Jersey:

Part I

The ProofPower System

THE METALANGUAGE

ProofPower has been implemented using Standard ML of New Jersey (SML-NJ), an implementation of Standard ML [4] developed by Lucent Technologies.

SML-NJ provides some extensions to the standard which have been used in the implementation of ProofPower.

These include:

- A means for storing the state of an ML session in a 'heap image'.
- Programmable control over the compiler's input and output streams.

These extensions have been used in the development of ProofPower to achieve the following effects:

- The theory hierarchy is held securely in what is called in the ProofPower documentation a 'theory database'.
- The character set accepted in the metalanguage (ML) has been extended by including appropriate logical and mathematical symbols. These extensions to the character set are mainly treated as new alphabetic characters and may therefore appear in ML variable names.
- Facilities have been provided for quotation of object language expressions and for automatic pretty-printing of values which represent such expressions.

To ensure that the ProofPower theory hierarchy is properly maintained, some of the effects normally obtained by commands supplied with SML-NJ should be obtained using alternative commands supplied with ProofPower. Users of ProofPower should only use the supplied ProofPower function instead of the corresponding SML-NJ function whenever they are operating on a ProofPower database.

ProofPower users should use the UNIX command **pp_make_database** instead of the SML-NJ commands **exportML** or **exportFN** to make a new database.

Some ProofPower ML commands supersede SML-NJ ones and we recommend you do not use the SML-NJ versions when working with ProofPower.

Use ProofPower Command	Instead of SML-NJ command
save_and_quit	**exportML**
quit	**OS.Process.exit**
use_file	**use**

Full details of the ProofPower versions of these commands may be found in the ProofPower *Reference Manual* [11].

ProofPower was originally developed for the 1990 version of Standard ML. SML-NJ implements the 1997 revision to the standard. For backwards compatibility, ProofPower provides many of the 1990 basis library functions in addition to the new standard basis library functions.

ProofPower CONTROLS

Some aspects of the behaviour of ProofPower are determined by the settings of various *controls* which the user of ProofPower may change.

There are three types of control, according to the type of the Standard ML value which exercises the control. The value may have type `bool`, `int` or `string`. Boolean controls are often referred to as "flags".

The procedures available for manipulating controls are shown in table 2.1.

name	**purpose**
get_controls	return all control values
get_flag	return value of a flag
get_flags	return all flag values
get_int_control	return an integer control value
get_int_controls	return all integer control values
get_string_control	return string control value
get_string_controls	return all string control values
new_int_control	create a new integer control
new_flag	create a new flag
new_string_control	create a new string control
reset_int_control	set an int control to its default value
reset_flag	set a flag to its default value
reset_string_control	set a string control to its default value
set_int_control	set an int control value
set_flag	set a flag
set_flags	set all flags
set_string_control	set a string control value
set_string_controls	set all string control values
reset_controls	set all control values to their defaults
reset_int_controls	set all int control values to their defaults
reset_flags	reset all flags to their default values
reset_string_controls	set all string control values to their defaults
set_controls	set all control values
set_int_controls	set all int control values
set_string_controls	set all string control values

Table 2.1: ML Procedures for Manipulating Control Values

name	default	purpose
ignore_warnings	false	Causes warnings to be ignored.
illformed_rewrite_warning	false	Causes output of a diagnostic when rewriting is inhibited due to bound variable capture.
pp_add_brackets	false	When set the pretty printer uses more brackets.
pp_let_as_lambda	false	If set *let*-expressions are not pretty printed as such.
pp_prefer_current_language	false	If this flag is false, then subterms are printed without changing language whenever possible. If the flag is true, then subterms are printed using the language of the current theory whenever possible.
pp_print_assumptions	true	If this flag is set then assumptions of theorems are printed in full. Otherwise a single dot is printed to indicate the presence of assumptions, but the assumptions themselves are not printed.
pp_show_HOL_types	false	Causes extensive inclusion of type information in the output if the pretty printer.
pp_show_index	false	When set, the pretty printer puts indexing brackets round the identifier occuring first in any term (used by the theory listers).
pp_types_on_binders	false	Causes the pretty printer to display type information on binding occurrences of variables.
pp_use_alias	true	When set the pretty printer will print terms using applicable aliases for the constants occuring in the terms.
profiling	false	Turns on the ProofPower profiling facilities.
resolution_diagnostics	false	Controls the output of diagnostics from the resolution package.
sorted_listings	false	Causes theory listings to be sorted according to keys
subgoal_package_quiet	false	This flag may be used to suppress all output from the subgoal package when running proofs.
subgoal_package_ti_context	true	Determines whether the type inference context is set by the subgoal package, causing type-checking of terms to be influenced by the types of free variables in the current subgoal.
ti_verbose	true	When set, diagnostics printed by the HOL type inferrer include the types of the variables in the failing term.
use_file_non_stop_mode	false	When set errors arising while processing a file using 'use_file' do not cause processing to be aborted.
use_extended_chars	true	When true extended characters are output while pretty printing, when false ascii keywords are output. Does not affect input or LaTeX output.
gc_messages	false	When true this turns on production of garbage collection progress messages by the Standard ML compilation system.

Table 2.2: Boolean Controls (Flags) for HOL and Z

name	default	purpose
check_is_z	false	Controls the checking of theorems by Z inference facilities.
standard_z_terms	false	Influences the checking of terms by the Z parser and type checker.
standard_z_paras	true	Influences the checking of paragraphs by Z parser and type checker.
z_allow_free_vars_in_axioms	false	When this and *z_use_axioms* are set, Z axiomatic descriptions are allowed to contain free variables. This is not standard Z behaviour, and is advised only for type-check only mode.
z_type_check_only	true	When set, certain aspects of the processing of Z are inhibited, giving a faster response for type-checking only.
z_use_axioms	true	When set, Z 'axiomatic descriptions' are entered as axioms (otherwise they are definitions with consistency caveats).
compactify_z_terms	true	When set, the type inferrer optimises the space occupied by the HOL types in the translation of Z terms into HOL. This space optimisation will typically reduce garbage-collection overheads, so it should normally be turned on.
subscript_z_schema_ops	false	When set, the schema operators that overload logical operators in standard Z must be distinguished with a subscript 's' (i.e., $\forall_s$, $\exists_s$, $\wedge_s$ etc. are used for the schema operators, and $\forall$, $\exists$, $\wedge$ etc. are reserved for the logical operators). In terms of the grammar in section 6.4.12 below, this means that the second alternative in the production for *Schema2* is disallowed as is the last alternative in the production for *SQuant*.
pp_quote_z_strings	false	When set, the Z pretty printer inserts Z quotation symbols around object language strings. This is for use in implementing object languages that include Z but with variant conventions for strings.

Table 2.3: Additional Boolean Controls (Flags) for Z

name	default	purpose
compactification_mask	0	Controls the compactification algorithms.
line_length	-	The number of characters per line to be used by the pretty printer. The default is the number of characters per line in the controlling terminal at the start of the session.
listing_indent	2	Used to control the formatting of theory listings.
pp_format_depth	~1	Influences the formatting produced by the pretty printer.
pp_top_level_depth	~1	Influences the formatting produced by the pretty printer.
RW_diagnostics	0	controls tracing from the Reader/Writer
thl_char_width	8	Used to control the formatting of theory listings.
thl_chars_per_tab	5	Used to control the formatting of theory listings.
thl_line_width	13	Used to control the formatting of theory listings.
thl_tab_width	50	Used to control the formatting of theory listings.
tactic_subgoal_warning	20	When the number of subgoals arising from a tactic application exceeds this value the subgoal package gives a warning before displaying the subgoals.
undo_buffer_length	12	Determines how many steps of a proof can be undone by the subgoal package (using ‘undo’).

Table 2.4: Integer Controls for HOL and Z

name	default	purpose
prompt1	`":) "`	This is the prompt used by ProofPower to invite you to type some input.
variant_suffix	`"’"`	This is the character used to decorate a variable name when alpha conversion takes place.

Table 2.5: String Controls for HOL and Z

FORMAT OF ProofPower DIAGNOSTICS

The ProofPower specification and proof tools (i.e., the program `pp`, and its language-specific variants `hol` and `zed`) report errors via the ML exception mechanism. The error messages have the following general format:

```
<<Optional Explanatory Text>>
Exception− <<Error Type>> ∗ <<Text>> [<<Function>>.<<Number>>] ∗ raised
```

Some recoverable error conditions are treated as warnings. Depending on the value of the system control flag *ignore_warnings* and on whether you are running interactively, functions detecting such conditions give you the option to ask them to recover from the error. The format of a warning message has the general form.

```
<<Optional Explanatory Text>>
*** WARNING <<Number>> raised by <<Function>>: <<Text>>
Do you wish to continue (y/n)?
```

For example, the following input, which contains a syntax error:

ProofPower Input
```
:> ⌜(if a then b else )c⌝;
```

produces the error message:

ProofPower Output
```
Syntax error in: ⌜ ( if a then b else <?> )
) is not expected after ⌜ ( if ... then ... else
Exception− Fail ∗ Syntax error [HOL−Parser.19000] ∗ raised
```

Most of the functions for manipulating syntax, and carrying out proofs, do not produce any optional explanatory text. For example, the following input, in which an inference rule is applied to a term with the wrong type:

ProofPower Input
```
:> asm_rule⌜1 + 2⌝;
```

produces the following error report:

ProofPower Output
```
Exception− Fail ∗ ⌜1 + 2⌝ is not of type ⌜:BOOL⌝ [asm_rule.3031] ∗ raised
```

As a final example, the most commonly seen warning condition is when you ask to delete an object from the current theory; for example, in a theory in which a number of constants and types have been defined, the input:

ProofPower Input

```
:> delete_type"REAL";
```

might produce the warning message:

ProofPower Output

```
*** WARNING 12012 raised by delete_type: Deletion of REAL would require the
 deletion of Constants:π, e; Type:REAL
Do you wish to continue (y/n)?
```

if you answer `y`, then *delete_type* will continue and delete the objects listed in the error message. If you answer `n`, it will raise an exception:

ProofPower Output

```
...
Do you wish to continue (y/n)? n
Exception− Fail * execution of delete_type abandoned [warn.10003] * raised
```

In more detail, the various components of an error message are as follows:

Optional Explanatory Text If present, this gives an extended description of the problem encountered. It is typically used by subsystems such as the parser or type inferrer, which use a custom layout for their error messages. If it is present then the *Text* part of the message is normally a very brief summary.

Error Type This may either be: *Fail* indicating an error which the detecting function cannot recover form but which may, in some circumstances, sensibly be handled by the calling function. *Error* indicating an unrecoverable error condition which should not be handled. Either the input is so seriously flawed that the condition should be reported to the user immediately, or an unexpected failure has occurred (and should be reported to ICL).

Text In the absence of the optional explanatory text, this is the main description of the error condition. The text of the message is generated from an entry in an error message database, together with any details specific to the current failure - for instance, the input causing the failure. The reference manual entry for the function raising the error may contain further details of the cause of the error.

Function This is the name of the function or subsystem raising the error.

Number This is the error number, which is an index into the error message database (c.f. *get_error_message*). The reference manual entry of the function raising the exception will usually contain the original text of the message, and sometimes a description of the causes of that particular error.

It is intended that errors raised following the above format will either give the user a pointer to the solution of the problem, or, if necessary, assist support staff understanding the problem. If you contact ICL concerning a problem with ProofPower, it will be helpful to have available the text of any relevant error messages. At least the function name and the error number should be noted.

If an exception is raised with a format other than the above then it usually indicates a Standard ML programming error. For example, trying to evaluate *1 div 0* will raise the Standard ML exception *Div*:

ProofPower Output
```
Exception− Div raised
```

If problems arise with files, such as warnings of missing files or unexpectedly being denied read permission, you should first contact your system administrator to see whether the problem is due to the local installation.

CONCRETE SYNTAX NOTATION

The notation used for grammars in this document is BNF following the relevant British Standard [5]. The productions in a grammar are usually presented interleaved with narrative text and distinguished by a vertical bar in the left margin headed with the letters BNF. Comments within rules are bracketed with ‘(∗’ and ‘∗)’. Defining occurrences of identifiers in grammars are usually shown in **bold type** and have an entry in the index to this document. The subset of BNF used may be summarised as follows:

A *grammar* consists of a non-empty list of what are referred to as productions. Each *production* in the grammar consists of: an identifier, referred to as the *non-terminal symbol* of the production, an equals sign, and a list of one or more alternatives followed by a semicolon. The production introduces (some of the) alternatives for the sentences of a *language* corresponding to the non-terminal symbol and determined by the grammar as a whole. Lists of alternatives are separated by vertical bars. The vertical bar denotes alternation (i.e. union of languages). An *alternative* is a comma-separated list of items. The comma denotes concatenation.

BNF

| **Grammar** = *Identifier*, ‘=‘, *Alternatives*, ‘;‘, {*Identifier*, ‘=‘, *Alternatives*, ‘;‘};
| **Alternatives** = *Alternative*, {‘|‘, *Alternative*};
| **Alternative** = *Item*, {‘,‘, *Item*};

An *item* is a primary optionally followed by a subtraction sign and another item. The subtraction sign denotes set-theoretic difference of languages. A *primary* is either a list of alternatives enclosed in one of square brackets or braces, or it is an atom. Square brackets denote optional omission. Braces denote repetition (zero or more occurrences).

BNF

| **Item** = *Primary*, [‘−‘, *Item*];
| **Primary** = ‘[‘, *Alternatives*, ‘]‘ | ‘{‘, *Alternatives*, ‘}‘ | *Atom*;

An *atom* is a list of (one or more) alternatives enclosed in round brackets, or it is a special symbol or a simple symbol. The round brackets are used for grouping: either for clarity or to allow a list of alternatives or an item to be used as a primary. A *special symbol* is some arbitrary text enclosed in question marks. It is used where it is not desired to give a more detailed formal description of a construct. A *simple symbol* is an identifier or a string. The identifier is said to be a *terminal symbol* if it does not appear on the left-hand side of any production in the grammar (and so denotes a language specified by other means, e.g. another grammar). A string denote the language comprising just that string.

BNF

| **Atom** = ‘(‘, *Alternatives*, ‘)‘ | *Special* | *Simple*;
| **Special** = ‘?‘, *Text*, ‘?‘;
| **Simple** = *Identifier* | *String*;

The terminal symbols of this grammar are informally defined as follows:

BNF

| **Text** = ? *Any string of characters not containing a question mark* ?;
| **Identifier** = ? *An alphanumeric identifier* ?;
| **String** = ? *A string of characters enclosed in quotation marks* ?;

Part II

ProofPower-HOL

ProofPower-HOL CONCRETE SYNTAX

In this chapter, the concrete representation of the ProofPower-HOL language is described.

5.1 Lexical Analysis

The following subsections describe the rules by which a fragment of HOL text, i.e. a sequence of what we shall call *lexical units*, is construed as a sequence of terminal symbols to be parsed according to the grammar defined in section 5.2.1.

The rules are intended to meet the following requirements:

1. names may have both alphanumeric and non-alphanumeric components and have components which are subscripted or superscripted;
2. spaces are not normally be required around frequently occuring names such as '∀', '$\exists_1$', '=' etc.;
3. the lexical status of names is under the control of the user rather than the system.

The approach taken generalises that taken in the lexis of Standard ML by allowing names to be formed by joining alphanumeric and non-alphanumeric components using underscores or special characters which indicate superscription or subscription. In addition, sequences of alphanumeric and non-alphanumeric characters may be declared to act as terminator symbols which do not require space separation from adjacent characters unless those characters are underscores or the superscription or subscription characters.

5.1.1 Lexical Units

To describe the lexical analysis of HOL we consider the HOL text to have been first separated into a sequence of lexical units (as part of the macro processing which embeds HOL in an extension of Standard ML). These lexical units may be though of as "extended characters" corresponding to one or more actual characters in the input stream.

A lexical unit is one of the following:

- A single character.
- A keyword symbol such as the superscription symbol, "⅄", or the subscription symbol "ϒ".
- A term quotation.
- A type quotation.
- A string literal. These follow the same rules as string literals in Standard ML.
- A character literal. These follow the same rules as string literals in Standard ML except that they are delimited by the backprime character '`'.
- A comment. These are delimited by "`(*`" and by a "`*)`" as in Standard ML.

Lexical units are grouped into categories as shown in table 5.1.

Thus symbolic units comprise the ASCII characters other than alphanumerics and the punctuation characters, the characters with character codes greater than 127 and the keyword symbols other than the superscription and subscription keywords.

Category	Contents
Punctuation	‘(’, ‘)’, ‘{’, }’, ‘[’, ‘]’, ‘:’, ‘;’, ‘,’, ‘|’, ‘•’ and ‘$’
Numeric	‘*0*’ . . . ‘*9*’
Alphanumeric	‘*A*’ . . . ‘*Z*’, ‘*a*’ . . . ‘*z*’, ‘*0*’ . . . ‘*9*’ and ‘′’
Copula	‘⅄’, ‘ϒ’ and ‘_’
Space	Space, tab, newline and other formatting characters and comment units
Literal	A string or character literal or a term or type quotation
Term Quotation	A term quotation
Type Quotation	A type quotation
Symbolic	Any lexical unit not in the one of the other categories

Table 5.1: Lexical Units for ProofPower-HOL

5.1.2 Control of Lexical Analysis and Parsing

ProofPower will allow the user to control the lexical analysis and parsing of ProofPower-HOL through the following forms of declaration (in which the term *name* denotes a sequence of one or more alphanumeric, copula or symbolic lexical units).

Binder Declarations These allow binder status to be associated with a name.

Fixity Declarations These allow infix, prefix or postfix status and a numeric precedence value to be associated with a name. They also allow the ordinary (“nonfix”) status to be restored to a name which has previously been given infix, prefix, postfix or binder status.

Terminator Declarations These allow a name to be given the status of a lexical terminator in the sense that, while such a name forms a name on its own, it may only form part of a longer name if it is tied to the other constituents of the name with copula characters (or, in fact, if it forms part of a longer terminator). The precise rules for forming identifiers are given in the next section. A terminator must begin with a symbolic lexical unit.

5.1.3 Lexical Analysis

Sequences of lexical units are classified into *lexemes* as follows:

Lexeme	Sequence of Lexical Units
Identifier	A sequence of one or more alphanumeric, symbolic or copula lexical units, which either has the form of a floating point number or in which (1) any subsequence which has been declared as a terminator is either adjacent only to copula units, or is a subsequence of a terminator, and (2) alphanumeric and symbolic units only appear adjacently within subsequences which have been declared as terminators. A floating point number comprises a sequence of digits giving the integer part, followed by a decimal point, optionally followed by a sequence of digits giving the mantissa, all optionally followed by a possibly negative exponent part.
Punctuation	A sequence comprising exactly one punctuation character
Space	A sequence comprising one or more space units
Literal	A sequence comprising exactly one literal unit
Term Quotation	A sequence comprising exactly one term quotation unit
Type Quotation	A sequence comprising exactly one type quotation unit

Table 5.2: ProofPower-HOL Lexemes

Thus the syntax of identifiers is as given by the following grammar:

BNF

```
Id          =   {Copula}, Atom, [{Copula, {Copula}, Atom}, {Copula}]
            |   Copula, {Copula}
            |   Num, {Num}, ‘.‘, {Num}, [(‘e‘ | ‘E‘), [‘∼‘], Num, {Num}];

Atom        =   AlNum , {AlNum}
            |   Sym, {Sym}
            |   Terminator;

Terminator  =  Sym, {AlNum | Sym | Copula};
```

in which *Num*, *AlNum*, *Copula* and *Sym* stand for numeric, alphanumeric, copula and symbolic lexical units respectively, and in which instances of *Terminators* are restricted to sequences of lexical units which have been declared as terminators.

A sequence of lexical units is *lexically analysed* by processing it from left to right at each stage extracting the longest possible lexeme. The sequence of lexemes is in error if any of the following holds

1. the last lexeme of the sequence is the ‘$’ punctuation character;
2. the sequence contains the ‘$’ punctuation character followed by a type or term quotation;
3. the sequence contains a character literal which does not contain exactly one character

The non-space lexemes in the sequence of lexemes resulting from lexical analysis are then processed from left to right to give terminal symbols in the grammar as indicated in the following table (in which, if two or more rows apply then the first applicable row should be used and in which some distinctions may depend on the context within the grammar within which the terminal symbol is required).

Terminal	Lexemes
‘`{`’, ‘`}`’, ‘`(`’, ‘`)`’, ‘`[`’, ‘`]`’, ‘•’, ‘`:`’, ‘`,`’, ‘`|`’, ‘`;`’	Punctuation
‘`and`’, ‘`else`’, ‘`if`’, ‘`in`’, ‘`let`’, ‘`then`’	Identifier
Name	Identifier or Literal or ‘`$`’ followed by any lexeme other than type or term quotation
Binder	Identifier declared as binder
InTmOp	Identifier declared as infix or the punctuation lexeme ‘`,`’
InTyOp	Identifier declared as infix
PostOp	Identifier declared as postfix
PreOp	Identifier declared as prefix
TmQ	Term Quotation
TyQ	Type Quotation

Table 5.3: Terminal Symbols for the HOL Grammar

Thus ‘`$`’ acts as an escape character and suppresses any special interpretation of what follows. ‘`$`’ followed by a string or character literal is taken to mean the HOL name represented by the contents of the literal — this might be used, for example, to give access from the standard HOL syntax described here to a language which allowed spaces in its identifiers or which had ‘`let`’ or ‘`if`’ etc. as identifiers.

5.1.3.1 Examples

Assume that ‘=’ and ‘+’ have been declared as infix and as terminators that ‘$\exists_1$’ has been declared as a binder and a terminator and that ‘*div*’ has been declared as infix.

Then no space is required after the ‘$\exists_1$’ or the ‘*v*’ in ‘$\exists_1\delta\bullet 1 div\delta = 1$’ or in ‘$\exists_1 x\bullet 1 + x = 1$’, however a space is required after the ‘*v*’, but not after the ‘$\exists_1$’ in ‘$\exists_1 x\bullet 1div\ x = 1$’; spaces are required in both places in ‘$\exists_1\ \_x\bullet 1div\ \_x = 1$’.

‘`$`$\exists_1$’ denotes the same function as ‘$\exists_1$’ but with ordinary function application syntax rather than the special binder syntax.

A name such as *‘=1’* may be declared as a terminator, and then *‘x=1’* and *‘=1x’* each contain two identifiers, whereas *‘x_=1’* is a single identifier. *div* could not be declared as a terminator since it does not begin with a symbolic character.

5.2 Concrete Syntax

The grammar of the ProofPower-HOL language is divided into a number of parts which build to form the whole language. These are given in the subsections of section 5.2.1. Section 5.2.2 provides additional explanatory material on various aspects of the grammar.

5.2.1 Collected Grammar

The grammar for HOL is given in sections 5.2.1.1-5.2.1.5 in terms of the terminal symbols described in section 5.1.1. The rules whereby a fragment of text is construed as a sequence of these terminal symbols are given in section 5.1.

Sections 5.2.1.1 and 5.2.1.2 give productions for HOL terms and types. Sections 5.2.1.3-5.2.1.5 give productions for certain auxiliary nonterminals used in the productions for terms.

5.2.1.1 Terms

To specify the concrete syntax for terms an ambiguous grammar is given, together with rules to say how any ambiguity in parsing a sentence should be resolved. The comments against the grammar productions refer to the sections where additional grammar rules and explanations are given.

BNF

```
Tm    =
          Binder, BndVars, ‘•‘, Tm                (* binding construct       5.2.1.4 *)
      |   ‘let‘, L, { ‘and‘, L }, ‘in‘, Tm        (* let clause              5.2.1.5 *)
      |   ‘if‘, Tm, ‘then‘, Tm, ‘else‘, Tm        (* conditional             5.2.2.3 *)
      |   OTm                                     (* operator term           5.2.2.5 *)
      |   Tm, ‘:‘, Ty                             (* type cast               5.2.1.2 *)
      |   Tm, Tm                                  (* application             5.2.2.4 *)
      |   Name                                    (* variable or constant    5.2.2.6 *)
      |   TmQ                                     (* term quotation          5.2.2.8 *)
      |   ‘{‘, OptTms, ‘}‘                        (* set display             5.2.2.2 *)
      |   ‘{‘, V, ‘|‘, Tm, ‘}‘                    (* set comprehension       5.2.2.2 *)
      |   ‘(‘, Tm, ‘)‘                            (* bracketted term         5.2.2.7 *)
      |   ‘[‘, OptTms, ‘]‘;                       (* list display            5.2.2.1 *)

OTm   =     (     Tm, InTmOp, Tm                  (* infix operation         5.2.2.5 *)
                | PreOp, Tm                       (* prefix operation        5.2.2.5 *)
                | Tm, PostOp )                    (* postfix operation       5.2.2.5 *)
OptTms    = [ Tm, { ‘;‘, Tm } ];
```

Ambiguities are to be resolved by the following rules:

1. The alternatives for *Tm* above are listed in order of increasing precedence, i.e. if a given sentence can be parsed according to two distinct alternatives then the alternative which comes earlier in the list above is to be preferred. For example, the sentence ‘$\lambda x \bullet t = u$’ would be parsed as ‘$\lambda x \bullet (t = u)$’ rather than as ‘$(\lambda x \bullet t) = u$’ by this rule. Note that this rule does not apply to the alternatives for *OTm*, which are disambiguated by a system of numeric precedences, see rule 3.

2. A term containing a type must be parsed so that each type extends as far to the right as possible. For example, ‘$t : \sigma_1\ \sigma_2 : \sigma_3$’ must be parsed so that the types it contains are ‘$\sigma_1\ \sigma_2$’ and ‘σ_3’ (not ‘σ_1’ and ‘σ_3’). This term might be rewritten as ‘$(t : \sigma_1\ \sigma_2) : \sigma_3$’ indicating that types ‘$\sigma_1\ \sigma_2$’ and ‘σ_3’ are equivalent, and that ‘t’ has that type.

3. Infix, prefix and postfix function applications (see section 5.2.2.5) associate according to their associated precedences. Higher values of precedence bind more tightly than lower values. Equal values of precedence associate to the right. Brackets may be used to force a different order of association. Defining the precedences of the standard HOL operators is outside the scope of this document.

4. Juxtaposition (see section 5.2.2.4) associates to the left. For example, if a sentence can be parsed as both '$(f\ t)\ u$' and '$f(t\ u)$', the former is preferred.

5.2.1.2 Types

Terms may be given a type explicitly using a 'type cast' of the form "*Tm*,':', *Ty*". This may be done for purely documentary purposes, or to force a less general type than would otherwise be inferred, or to disambiguate the use of aliases.

The syntax for types is given by the nonterminal *Ty* in the following grammar. Note that the grammar for types is ambiguous, but the text that follows shows how to disambiguate it.

BNF

```
Ty        =    Name
          |    TyQ
          |    Typars, Name
          |    Ty, InTyOp, Ty
          |    '(', Ty, ')';

Typars    =    Ty
          |    '(', Ty, { ',', Ty }, ')';
```

Type variables and nullary type constructors are included under the first alternative, *Name*, here; the name is construed as a type variable name if it begins with a prime ('′') character. In general a type is either such a name, or a quotation (see section 5.2.2.8), or a type constructor applied to a list of types.

Non-nullary type constructors are normally used in a postfix format. If a Binary type constructor is declared as "infix" then it should be used infix unless preceded by '$'. If a type constructor has lexical status "nonfix" then it is written postfix. If a non-nullary type constructor is given any other lexical status then the lexical status must be suspended using '$' before it can be used, and it will then be usable only as a postfix operator. If a type constructor takes more than one argument, the arguments are given as a comma-separated list enclosed in brackets. Brackets may also be used in more general ways, e.g.: to group the arguments of infix type constructors.

The binary type constructors for product, sum, and function (written as '$\times$', '$+$' and '$\rightarrow$' respectively) are typical examples of infix type constructors.

Ambiguities in the grammar for types are resolved using the following rules:

1. The postfix form is of higher precedence than the infix.

2. Infix type constructors are assigned a precedence when their infix status is declared (see section 5.1.2). Higher values of precedence bind more tightly than lower values. Equal values of precedence associate to the right.

5.2.1.3 Variable Structures

‘Variable Structures’ (often abbreviated as ‘varstructs’) may be used instead of simple variables in any variable binding construct where the variable to be bound would otherwise have a type formed by the ordered pair type constructor, ‘×’. In these circumstances it is convenient to allow distinct names to be bound to each element of the pair, so that these names may be used in the scope of the binding instead of expressions involving the projection functions from pairs (*Fst*,*Snd*). This effect is achieved by writing the variable names in the declaration part of the variable binding construct combined using the pair constructor ‘,’ and enclosed in brackets. The effect may be iterated, yielding a variable structure of arbitrarily nested pairs (consistent with type contraints).

A varstruct or any of its subterms may be given a type cast.

Examples of this notation are:

HOL terms
```
│        ⌜λ (a,b):(ℕ × ℕ) • f(a,b)⌝;
│        ⌜λ (a:ℕ,(b,(c,d),e)) •  f(a,b,c,d,e)⌝;
```

This precise syntax for varstructs is described by the following BNF syntax rule for the nonterminal *V*.

BNF
```
│        V     =     V1, [ ‘:‘, Ty ];
│        V1    =     Name
│              |     ‘(‘, V, { ‘,‘, V }, ‘)‘;
```

Bindings composed using varstructs are often called ‘paired abstractions’.

Interpretation

Varstructs can only be interpreted by giving the meaning of the variable binding construct (in this case a ‘paired abstraction’) in which they occur.

A paired λ-term is a syntactic abbreviation for (i.e. is parsed into the same term as) an application of the constant *Uncurry* to a simple λ-abstraction formed by abstracting over each of the constituent variables in turn. The function *Uncurry* converts a (so-called ‘curried’) function of two arguments into a function of a single argument (which must be a pair).

For example: ⌜λ (*a*, *b*) • *t*⌝ yields the same term as ⌜*Uncurry* (λ*a*•λ*b*•*t*)⌝.

Where a binding construct is formed using a constant declared as a binder, the same effect takes place on the underlying λ-term.

E.g. ⌜∀ (*a*, *b*) • *p*⌝ yields the same term as ⌜$∀ (*Uncurry* (λ*a*•λ*b*•*p*))⌝.

Similar effects apply to other constructs which involve variable binding, such as let clauses and set abstractions.

⌜*let* (*a*,*b*) = *c in t*⌝ yields the same term as ⌜*Let*(*Uncurry*(λ (*a*, *b*) • *t*))*c*⌝ (cf. section 5.2.1.5).

5.2.1.4 Binder Abbreviation

Where λ-terms or nested terms formed by application of a constant (or variable) with lexical status of ‘binder’ appear, the inner occurrences of ‘λ’ or of the repeated binder may be omitted. This is also permitted when varstructs rather than simple variables are used in the bindings. This facility is not available for set abstractions.

Since a type in a cast applied to a varstruct extends as far as possble to the right, it may be necessary in some cases to add additional punctuation to delimit the type. This may be done with brackets or by the use of a semicolon. Where only the last of a series of variables has a type cast applied to it, the cast is deemed to apply to all variables preceding it up to the first which either has a type cast or is followed by a semi-colon.

HOL terms

$\ulcorner\forall\ x\ y\ z\ x_1\ y_1 \bullet f(x,\ y,\ z,\ x_1,\ y_1)\urcorner;$
$\ulcorner\forall\ x : BOOL \bullet x \lor \neg\ x\urcorner;$
$\ulcorner\forall\ (x,y) \bullet P\ x\ y\urcorner;$
$\ulcorner\forall\ x\ y : BOOL; \bullet x \Leftrightarrow y \lor x \lor y\urcorner;$
$\ulcorner\forall\ x\ y : BOOL;\ z\ x_1 : \mathbb{N} \bullet T\urcorner;$
$\ulcorner\forall\ (x : BOOL,\ y :) \bullet T\urcorner;$
$\ulcorner\forall\ (x,y) : (BOOL \times BOOL) \bullet T\urcorner;$
$\ulcorner\forall\ (x,y)\ ((z,x_1),y_1) \bullet T\urcorner;$
$\ulcorner\forall\ (x,y)\ (z,x_1)\ (y_1,z_1) : BOOL \times BOOL;\ x_2\ y_2;\ z_2 : \mathbb{N} \bullet T\urcorner;$
$\ulcorner\forall\ x : BOOL\ SET \bullet x \subseteq \{T,F\}\urcorner;$
$\ulcorner\forall\ (x : BOOL)\ SET \bullet \neg\ (SET = SET) \Rightarrow x\urcorner;$
$\ulcorner\forall\ x : BOOL;\ SET \bullet \neg\ (SET = SET) \Rightarrow x\urcorner;$

Note that the effect of the second semicolon in the last example is to stop the type constraint on z_2 applying to x_2 and y_2. However, in the first example, where there is no cast, the variables need not have the same type.

The intention of the grammar of binders is that the binder ranges over a series of blocks of bound variables, which are given by rule *BndVars* below. Each block may declare many names or many paired abstractions and is optionally typed. The blocks are separated by semicolons.

BNF

BndVars	=	*Block*, {‘;‘, *Block* };
Block	=	*V1*, { *V1* }, [‘:‘, *Ty*];

Interpretation

The simple λ-abstraction $\ulcorner\lambda x \bullet t\urcorner$ yields a term formed by the primitive ‘*mk_abs*’ constructor with the terms x and t as its arguments. All variable binding constructs in the ProofPower-HOL concrete syntax are syntactic abbreviations for less concise expressions in which all variable binding is done by simple λ-expressions.

The following four ProofPower-HOL fragments yield the same ProofPower-HOL TERMs.

HOL terms

$\ulcorner\forall\ x_1\ x_2\ x_3\ x_4 \bullet T\urcorner;$
$\ulcorner\forall\ x_1 \bullet \forall\ x_2 \bullet \forall\ x_3 \bullet \forall\ x_4 \bullet T\urcorner;$
$\ulcorner\forall\ x_1 \bullet (\forall\ x_2 \bullet (\forall\ x_3 \bullet (\forall\ x_4 \bullet T)))\urcorner;$
$\ulcorner(\$\ \forall)(\lambda\ x_1 \bullet ((\$\ \forall)\ \lambda\ x_2 \bullet ((\$\ \forall)\ \lambda\ x_3 \bullet ((\$\ \forall)\ \lambda\ x_4 \bullet T))))\urcorner;$

The use of ‘\$’ in the above to suppress the binder status of the name[1] ‘∀’ is discussed in section 5.1.3.

[1]Note that giving binder status to a name affects all uses of that name, whether as a constant or as a variable.

5.2.1.5 Local Definitions

A **let** clause assigns a local name to stand for the value of an arbitrary HOL term within the body of the clause. Where the value to be assigned to the local variable is a function then the value may either be expressed as a λ-expression. Alternatively the function may be defined in the **let** clause by an equation in which the function is applied on the left of the equation to one or more variable structures which serve as formal parameters to the function. The variables occurring in the varstructs are binding occurrences whose scope is the right hand side of the equation.

A single **let** clause may introduce several local definitions in parallel, separated by the terminal 'and'.

Examples of local definitions are:

HOL terms

$\ulcorner let\ (a,b) = c\ in\ f(a)\urcorner;$
$\ulcorner let\ (a{:}\mathbb{N},(b,(c,d),e)) = exp\ in\ f(a,b,c,d,e)\urcorner;$
$\ulcorner let\ x = exp1\ in\ exp2\urcorner;$
$\ulcorner let\ f\ x = exp1\ in\ exp2\urcorner;$
$\ulcorner let\ f\ (x_1,x_2)\ (y_1,(y_2,y_3),y_4) = exp1\ in\ exp2\urcorner;$
$\ulcorner let\ x = exp1$
$and\ f\ x = exp2$
$and\ g\ f\ (x,b)\ (y_1,(y_2,y_3),y_4) = exp3$
$in\ exp3\urcorner;$

For a local function definition the function name is followed by formal argument patterns, a sequence of variable structures in the same form as would be permitted in a 'λ'-expression. Production *L* below gives the part of the clauses between the **let** or **and** keywords and the 'in'.

BNF

L	=	*Vs* '=', *Tm*;	
Vs	=	*V*	| *Name*, *V1s*;
V1s	=	*V1*	| *V1s*, *V1*;

Thus if there is a list of varstructs after the **let** or **and** the first one must just be a name.

Interpretation

A **let** clause abbreviates the application of the constant *Let* to a lambda abstraction.

The following pairs of expressions yield the same ProofPower-HOL TERMs.

Identical HOL TERMs

$\ulcorner let\ x = t\ in\ u\urcorner;$
$\ulcorner Let(\lambda\ x \bullet u)\ t\urcorner;$

Identical HOL TERMs

$\ulcorner let\ x = t\ and\ y = u\ and\ z = v\ in\ w\urcorner;$
$\ulcorner Let(Let(Let(\lambda\ x\ y\ z \bullet w)t)u)v\urcorner;$

Identical HOL TERMs

$\ulcorner let\ x = t\ in\ let\ y = u\ in\ let\ z = v\ in\ w\urcorner;$
$\ulcorner Let(\lambda\ x \bullet Let(\lambda\ y \bullet Let(\lambda\ z \bullet w)v)u)t\urcorner;$

Note that the defining expressions on the right hand side of the equations of the let clause are not in the scope of the variable binding created. This applies to all the expressions involved in multiple local declarations formed using **and** in the **let** clause.

Thus in : $\ulcorner let\ f\ x = f\ x\ in\ b\urcorner$ the local definition is not recursive since the second occurrence of f is not in the scope of the local declaration.

5.2.2 Explanations of the Grammar

The above sections gave the full grammar of ProofPower-HOL and the description of parts of it. The next sections complete the description.

5.2.2.1 Lists

The square bracket notation is for lists. Elements of lists are separated by semicolons, and the empty list is allowed.

Interpretation

The list '$[t;\ u;\ v\ \ldots]$' stands for '$Cons\ t(Cons\ u(Cons\ v(\ldots Nil)\ldots)))$', and the empty list '[ ]' stands for '*Nil*.'

5.2.2.2 Set Terms

Set terms may be given in two forms. Either by enumeration of the values, or by set comprehension. Examples of set terms are as follows.

HOL terms

```
│     ⌜{1; 4; 9; 16; 55}⌝;
│     ⌜{x | ∃ y : ℕ • y ≤ 5 ∧ x = y ∗ y}⌝;
│     ⌜{(x,y) | x ≥ y}⌝;
```

Interpretation

The set '$\{t;\ u;\ v\ \ldots\}$' stands for '$Insert\ t(Insert\ u(Insert\ v(\ldots Empty)\ldots)))$', and the empty set '{ }' stands for '*Empty*.'

A set comprehension stands for the constant *SetComp* applied to a lambda term. The set '$\{x|p\}$' stands for '$SetComp(\lambda x \bullet p)$.'

5.2.2.3 Conditionals

Conditionals are of the form '*if* condition *then* consequence *else* alternative.'

Interpretation

The conditional form stands for an application of the polymorphic constant '*Cond*'. The conditional '*if* t *then* u *else* v' stands for the term '$Cond\ \ t\ \ u\ \ v$.'

5.2.2.4 Juxtaposition

Juxtaposition stands for the application of a function to an argument, e.g. *f a* stands for application of *f* to *a*. This corresponds to the primitive *mk_comb*.

5.2.2.5 Infix, Prefix and Postfix Operators

As described in section 5.1.2 the user may declare that a name[2] is to be used with infix, prefix or postfix syntax. A numeric precedence is assigned to the name in such a declaration and is used to determine the order of application, as described in rule 3 of section 5.2.1.1, and illustrated by the following examples.

Assuming that '*I1*', '*I4*', '*R2*', '*R5*', '*S3*' and '*S6*' are two infix, two prefix and two postfix operators respectively with precedences as given in the name (i.e., precedences of 1, 4, 2, 5, 3 and 6 respectively) then the brackets in the following examples are redundant.

HOL fragments

```
(t I4 u) I1 v        t I1 (u I4 v)        (t I4 u) I4 v
R2 (R5 t)            R5 (R2 t)
(t S6) S3            (t S3) S6
(R5 t) S3            R2 (t S6)
(R5 t) I1 (u S6)     R2 ((t I4 u) S3)
```

Interpretation

These forms simply provide additional ways of expressing function application. For example, in each of the following three examples the two terms shown are equivalent. In these examples, we use the '$' symbol to suppress the infix, prefix or postfix status of an identifier (as described in section 5.1).

Equivalent HOL terms

```
l   i_op   r
($i_op   l)   r
```

Equivalent HOL terms

```
pre_op   a
$pre_op   a
```

Equivalent HOL terms

```
a   post_op
$post_op   a
```

5.2.2.6 Names of Variables and Constants

This alternative is for variables and constants, it includes any identifier which has not been declared to have special syntactic status (as defined in section 5.1.2) and which is not escaped by preceding it with a '$'.

For example with typical binder declarations, '$∀' the name of the universal quantification operation considered as a constant in its own right, whereas '∀' is not a name.

[2]Note that operator names need not necessarily be constants; they may be variables with associated fixity syntax.

5.2.2.7 Grouping

Brackets may be used to override the precedence and precedence rules.

5.2.2.8 Quotation

ProofPower is a multilingual system which supports mixed language working.

The lexical terminals '*TmQ*' and '*TyQ*' in the grammar of ProofPower-HOL represent points at which TERMs or TYPEs respectively may be entered by nested quotation in some other language.

5.3 Aliases and Overloading

The HOL system allows *aliases*, to be declared for particular instances of constants. This facility allows different names (and associated fixities and precedences) to be used for the same function. For example, the symbol '⇔' is set up in the supplied system as an alias for the boolean instance of the polymorphic equality operator, '='. '⇔' is given a lower precedence than '=', so that a term such as $\ulcorner x = y \Leftrightarrow a = b \urcorner$ associates as $\ulcorner (x = y) \Leftrightarrow (a = b) \urcorner$.

Different constants may have a common alias. This allows aliases to be used to achieve the effect of overloaded constant names. For example, it would be natural to use '+' as an alias for the additions on natural number, integers, rational etc.

An alias may be the same as a constant name. This allows an application in which the terminology conflicts with that used in a library theory to be handled in its own terms. For example, if `sin` denotes the sine function in a library theory, the name `sin` might be made an alias of `transgression` in a theory of ethics.

5.4 Type Abbreviations

To ease the use of complicated types, a name (with a possibly empty list of *type parameters*) may be declared as a *type abbreviation*. A name which has been declared as a type abbreviation may be used either as a *Name* in the production for *Ty* in section 5.2.1.2 above, or, if an infix declaration for the name is in force, as an *InTyOp*.

For example, if the name *auto*, with the single parameter $'a$ had been declared as an abbreviation for the function type $'a \rightarrow 'a$. Then $(bool)auto$ would denote the same type as $bool \rightarrow bool$.

Part III

ProofPower-Z

ProofPower-Z CONCRETE SYNTAX

6.1 STANDARD AND EXTENDED Z

The ProofPower-Z language is an extension of an approximation to the Z language of the ongoing standardisation activity for Z. In this document the term *extended Z* is used to refer to the full ProofPower-Z language, and the term *standard Z* for the subset which approximates the language of the standard.

Normally, the ProofPower parser and type checker prohibit the use of non-standard features within the paragraphs of a specification but allow them within predicates and expressions entered as term quotations (see 6.2 and 6.4 below). (This default behaviour may be modified by changing the settings of the system control flags *standard_z_paras* and *standard_z_terms*.)

The main differences between standard and extended Z are as follows;

- Extended Z is higher-order, in effect the ProofPower-HOL type, ⌜:𝔹⌝, of truth values acts as a given set, so removing the distinction between the syntactic categories of predicates and expressions in standard Z. In particular, in extended Z, one can state and prove theorems involving propositional variables.
- Nested term quotations are allowed in extended Z, permitting mixed language working (see section 6.3 below).
- Extended Z supports two cast operators required to enter fragments of Z during proof (see section 6.5.1 below).

(Remark, the first of these differences is not fully checked in the current implementation of the system.)

6.2 ENTERING A Z SPECIFICATION

A ProofPower-Z specification is presented as a LaTeX document using the facilities described in [6] and in what follows familiarity is assumed with the use of the extended character set discussed in that document.

The lexical rules for ProofPower-Z are defined in section 6.3 in terms of the slightly abstract view of the Z character set shown in table 6.1. Each of these characters be entered into a ProofPower-Z document or into the ProofPower system using a single key-stroke with the following exceptions:

- The guillemet characters, "≪" and "≫", are entered as `%<<%` and `%>>%`.
- A calligraphic letter such as "$\mathcal{M}$" that has not been assigned a single key-stroke is entered as `%calM%`.
- Most of the additional symbols from the LaTeX manual mentioned under *Symbol* are entered by enclosing the alphabetic part of the LaTeX name in percent characters. E.g. the infinity symbol, "∞", which is invoked by the macro `\infty` in LaTeX, is entered as `%infty%`. There are some exceptions to this rule, mainly because LaTeX has more than one name for some of the symbols. The exceptions are listed in the file `sievekeyword` provided in the subdirectory `sun3bin` or `sun4bin` of the installation directory.
- A character is entered as a subscript by preceding it with the character ⋎.

- The *Box* characters are entered simply by laying the box out as formal Z material in the sense of [6]. Note that the paragraph forms which do not use a box, namely, fixity paragraphs, given set definitions, abbreviation definitions and free type definitions, must be preceded by a line containing the two characters "Ⓢ*Z*" and followed by a line containing the single character "■". Boxes beginning with "┌" or "╒" must be terminated with a line beginning "└" (not "■").

The paragraph forms behave like ML functions executed at the top-level for their side-effects (and the formatting directive characters which terminate the boxes implicitly insert the semi-colon required to cause the function to be executed by the ML compiler after them).

Z term quotations, see section 6.4 below, are entered bracketed with the characters $\ulcorner_{Z}$ and $\urcorner$, for example, $\ulcorner_{Z}\langle 1, 2, 3\rangle\urcorner$. Z term quotations behave like ML functions executed for their result value.

As described in [6], the normal effect of the characters ↗ and ↕ is to begin and end sections of superscripted text. Thus, for example, the name of the iteration operation is "_ ↗ _ ↕", so that, something entered as *R* ↗ *i* ↕ prints as R^{i}.

Letter	A	B	C	D	E	F	G	H	I	J	K	L
	M	N	O	P	Q	R	S	T	U	V	W	X
	Y	Z										
	a	b	c	d	e	f	g	h	i	j	k	l
	m	n	o	p	q	r	s	t	u	v	w	x
	y	z										
GreekLetter	α	β	γ	δ	ϵ	ζ	η	θ	ι	κ	λ	μ
	ν	ξ		π	ρ	σ	τ	υ	ϕ	χ	ψ	ω
			Γ	Δ				Θ			Λ	
		Ξ		Π		Σ		Υ	Φ		Ψ	Ω
Digit	0	1	2	3	4	5	6	7	8	9		
Symbol	$\cup$	$\cap$	$\bigcup$	$\bigcap$	$\subseteq$	$\subset$	$\varnothing$	$\notin$	⦇	⦈	⩤	⩥
	$\lhd$	$\rhd$	$\vdash$	⨾	$\oplus$	⇸	$\rightarrow$	⤔	↣	⤀	↠	⤖
	$\leftrightarrow$	$\leq$	$\geq$	$<$	$>$	⁀	⁀/	$\neq$	$+$	$-$	$*$	#
	.	$\sim$	$\uplus$	⟦	⟧	/	\	_	$\wedge$	$\vee$	$\neg$	$\Rightarrow$
	$\Leftrightarrow$	$=$	$\in$	$\forall$	$\exists$	$\times$	&	?⊢	⨾	↾	$\overset{\oplus}{\oplus}$	
	and the calligraphic letters and any other symbols from tables 3.4, 3.5, 3.6 and 3.7 of the LaTeX User's Guide and Reference Manual, [3]											
Stop	,	;	:	•	(	)	[	]	{	}	$\langle$	$\rangle$
	⟦	⟧	$\ll$	$\gg$	$\vert$	::=	$\hat{=}$	$\mathbb{B}$	$\mathbb{C}$	$\mathbb{F}$	$\mathbb{N}$	$\mathbb{P}$
	$\mathbb{Q}$	$\mathbb{R}$	$\mathbb{S}$	$\mathbb{U}$	$\mathbb{Z}$							
Underscore	_											
Stroke	′	?	!									
Subscript	Subscripted forms of any of the above characters.											
Shift	↗	↕										
Box	AX	SCH	END	IS	ST	BAR						
Quote	"											
Format	A format character such as space, tab, line-break or page-break.											

Table 6.1: Character Set

6.3 LEXICAL ANALYSIS

Token A *token* is a sequence of characters, as shown in table 6.1, conforming to the grammar given in this section. The terminal symbols of the grammar are the sets of characters defined in the table, and the sentence symbol is *Token*. The different sorts of token correspond to the sorts of terminal symbols of the grammar for Z given in section 6.4, together with an extra sort of space tokens.

A sequence of characters is interpreted as a sequence of non-space tokens by a left-to-right scan taking tokens which are as long as possible and then discarding any *Space* tokens. If it is not possible to do this then the sequence of characters is erroneous.

ProofPower-Z supports inline comments within both paragraphs and terms. These comments follow the same rules as for comments in ProofPower-HOL and ProofPower-ML.

As an extension to Z, quotation of terms using other ProofPower languages, e.g. HOL, is allowed.

BNF

```
Token            =     Identifier
                 |     Decor
                 |     Narrative
                 |     Natural
                 |     Float
                 |     String
                 |     Punctuation
                 |     Quotation
                 |     Space;
```

Identifier There are four sorts of identifier.

BNF

```
Identifier       =     Alphanumeric
                 |     Greek
                 |     Symbolic
                 |     QuotedIdentifier;

Alphanumeric
                 =     Letter, {Letter | Digit | ‘_‘}, {Subscript}
                 |     ‘_‘, (Letter | Digit | ‘_‘),
                       {Letter | Digit | ‘_‘}, {Subscript};

Greek            =     GreekLetter, {Subscript};

Symbolic         =     (Symbol | Shift), {(Symbol | Shift)}, {Subscript}
                 |     Punctuation, Subscript, {Subscript};

QuotedIdentifier
                 =     ‘$‘, String;
```

Decoration Decoration comprises just a sequence of stroke characters

BNF

Decor = *Stroke*, {*Stroke*};

Numbers An natural number literal is a non-empty sequence of decimal digits

BNF

Natural = *Digit*, {*Digit*};

A floating point numeric literal comprises two non-empty sequences of digits separated by a decimal point together with an optional exponent part.

BNF

Float = *Digit*, {*Digit*}, ‘.‘, *Digit*, {*Digit*}, [(‘*e*‘ | ‘*E*‘), [‘~‘], *Digit*, {*Digit*}];

A token preceded by the reserved identifer ‘.’ is not interpreted as a floating point numeric literal. E.g., in ‘$(x, y, (a, b)).1.2$’, ‘1.2’ is treated as two natural number literals separated by a ‘.’ and not a floating point literal so that the expression is a valid tuple component selection (see section 6.4.15).

Strings A string literal denotes a sequence of arbitary text. String literals conform to the same syntax as HOL string literals, which follow Standard ML.

BNF

String = ‘"‘, {*StringItems*}, ‘"‘;
StringItems = ? *As in Standard ML* ?;

Narrative The means for delimiting the narrative sections between formal material in a Z document is defined in [6]:

BNF

Narrative = ? *See Document Preparation User Guide* ?

Punctuation This kind of token includes the stop and box characters of section 6.2 symbols and an underscore optionally followed by a question mark or an exclamation mark.

BNF

Punctuation = *Stop*
| *Box*
| ‘_‘
| ‘_?‘
| ‘_!‘;

Quotation A quotation must be a term valued quotation . E.g. an HOL type quotation is not allowed, while an HOL term quotation is.

BNF

Quotation = ? *A term valued quotation* ?;

Space A space token is a sequence of one or more white space characters or a comment.

BNF

```
|    Space        =    Format, {Format}
|                 |    Comment;
```

Comments follow the syntax for comments in Standard ML:

BNF

```
|    Comment      =    ? As in Standard ML ?;
```

6.4 GRAMMAR

The following grammar defines two languages over the set of all non-space tokens as defined in section 6.3. The languages are the language of Z specifications corresponding to the non-terminal *Specification* in section 6.4.1 and the language of Z terms corresponding to the non-terminal *Term* in section 6.4.3. The language of terms is, in a sense, an extension to standard Z permitting entry of Z constructs needed as parameters to proof procedures etc.

The specific terminal symbols of the grammar are the punctuation symbols and reserved identifiers listed below.

Punctuation

‘$\mathbb{P}$‘	‘:‘	‘;‘	‘\‘	‘(‘	‘)‘	‘[‘	‘]‘	‘{‘	‘}‘	‘$\langle$‘	‘$\rangle$‘
‘$\ll$‘	‘$\gg$‘	‘$\bullet$‘	‘_‘	‘::=‘	‘AX‘	‘SCH‘	‘END‘	‘IS‘	‘ST‘	‘BAR‘	

Reserved Identifiers

‘$\exists_1$‘	‘θ‘	‘λ‘	‘μ‘	‘Δ‘	‘Ξ‘	‘.‘	‘...‘	‘>>‘	‘$\Leftrightarrow$‘	‘$\wedge$‘	‘$\vee$‘
‘$\neg$‘	‘$\Rightarrow$‘	‘$\forall$‘	‘$\exists$‘	‘$\mid$‘	‘$\times$‘	‘&‘	‘,‘	‘/‘	‘$\in$‘	‘=‘	‘$\hat{=}$‘
‘==‘	‘?$\vdash$‘	‘$\upharpoonright_s$‘	‘$⨟_s$‘	‘$\backslash_s$‘	‘$\Leftrightarrow_s$‘	‘$\wedge_s$‘	‘$\vee_s$‘	‘$\neg_s$‘	‘$\Rightarrow_s$‘	‘$\forall_s$‘	‘$\exists_s$‘
‘*false*‘	‘*pre*‘	‘*true*‘	‘*let*‘								

In addition to the reserved identifiers, the following identifiers associated with fixity paragraphs are mentioned in the grammar. These identifiers may be used as ordinary identifiers except in the appropriate position in a fixity paragraph.

Reserved Identifiers

‘*fun*‘	‘*function*‘
‘*gen*‘	‘*generic*‘
‘*rel*‘	‘*relation*‘
‘*leftassoc*‘	
‘*rightassoc*‘	

The only difference between punctuation symbols and reserved identifiers is that the former are not identifiers according to the lexical rules given in section 6.3 whereas the latter are. The grammar prohibits attempts to use the reserved identifiers as variables.

The general terminal symbols of the grammar are as shown in table 6.2.

Symbol	Description
Id	Identifier other than the reserved identifiers
Decor	A sequence of decoration characters
Natural	Natural number literal
Float	Numeric literal
Character	Character literal
String	String literal
Narrative	Informal text

Table 6.2: Terminal Symbols for the Z Grammar

6.4.1 Specification

A specification comprises a sequence of *paragraphs* interleaved with narrative text

BNF

```
|        Specification =      [Narrative], {Paragraph, Narrative}, [Paragraph];
```

6.4.2 Paragraphs

A paragraph takes one of 7 forms:

BNF

```
|        Paragraph    =       Fixity
|                     |       GivenSet
|                     |       AbbDef
|                     |       FreeTypeDef
|                     |       AxBox
|                     |       Constraint
|                     |       Conjecture;
```

6.4.3 Term

This non-terminal is an extension to Z enabling expressions, predicates and schemas to be supplied as parameters to proof procedures etc.

A term comprises a Z predicate (which, in extended Z, includes expression and schema) together with optional generic parameters.

BNF

```
|        Term  =      [GenFormals], Pred;
```

Note that the above is actually the same as the *Constraint* paragraph. No ambiguity arises because of the use of different beginning and end markers for the two forms as described in section 6.2 above.

6.4.4 Fixity Paragraph

A fixity paragraph describes syntactic abbreviations which are to be used in the specification. The constructs which can be abbreviated are application of a function to a tuple of arguments, explicit instantiation of a generic constant and the membership predicate. These three possibilities are given by the *Function*, *Generic* and *Relation* options in the following production.

BNF

```
|        Fixity       =       Function , [Prec, [Assoc]], Template, {‘,‘, Template}
|                     |       Generic, [Prec, [Assoc]], GTemplate, {‘,‘, Template}
|                     |       Relation, Template, {‘,‘, Template};
```

Each of the three alternatives for a fixity paragraph begins with an identifier as shown in the following productions. The options in each of these productions support both the original ProofPower-Z syntax and the Standard Z syntax. These identifiers are not treated as reserved except when they appear at the beginning of a Z paragraph.

BNF

Function	=	‘*fun*‘ | ‘*function*‘;
Generic	=	‘*gen*‘ | ‘*generic*‘;
Relation	=	‘*rel*‘ | ‘*relation*‘;

A numeric precedence may be specified in a function or generic fixity paragraph. The precedence gives a non-negative numeric precedence for the abbreviations being described. Omitting the precedence number is equivalent to supplying it as *0*. This is only relevant to postfix, prefix, or infix templates, i.e., those that begin with the placeholder ‘_‘, end with the placeholder ‘_‘, or both begin and end with the placeholder ‘_‘ (respectively).

Prec	=	*Natural*;

Left or right associativity may be specified in a function or generic fixity paragraph using the identifiers listed in the following production. This is only relevant to infix templates, i.e., those that begin and end with the placeholder ‘_‘. If the associativity specification is omitted, then an infix template is treated as right associative, An example of a function defined with a right associative template is the function arrow in the Z toolkit, for which ‘$X \rightarrow Y \rightarrow Z$’ means the same as ‘$X \rightarrow (Y \rightarrow Z)$’ rather than ‘$(X \rightarrow Y) \rightarrow Z$’.

BNF

Assoc	=	‘*leftassoc*‘ | ‘*rightassoc*‘;

A template has the form of a sample use of the abbreviation with stubs for the arguments. The stubs are either ‘_’ (optionally followed by ‘?’ or ‘!’), corresponding to an argument position where a single expression is expected, or ‘...’, corresponding to an argument position requiring a list of expressions (a possibility which does not arise in generic fixity paragraphs or at the beginning and end of function and relation fixity paragraphs). The form ‘, ,’ may also be used and is equivalent to ‘...’. Quoted identifiers are not allowed in templates.

BNF

Template	=	([*Stub1*], {*Id*, *Stub*}, *Id*, [*Stub1*]) − *Id*;
GTemplate	=	([*Stub1*], {*Id*, *Stub1*}, *Id*, [*Stub1*]) − *Id*;
Stub1	=	‘_‘ | ‘_?‘ | ‘_!‘;
Stub	=	*Stub1* | ‘...‘ | ‘,,‘;

In standard Z, the arguments of functions, relations and generic operators must be expressions. In ProofPower-Z, functions and relations can also take predicates as arguments. The stubs ‘_?’ and ‘_!’ are for use in defining operators that use this extension: ‘_?’ indicates an argument that is to be parsed as a predicate and ‘_!’ indicates an argument that is to be parsed as a predicate or as an expression according to the context. See sections 6.5.1 and 6.5.2 below for examples.

The syntactic abbreviations introduced by a fixity paragraph are in force throughout the entire specification containing them. The following rules apply to the identifiers which appear in a template:

1. the first and last identifiers in the template must not appear anywhere in any other template in any fixity paragraph in the specification, unless that template introduces exactly the same syntactic abbreviation.

2. identifiers other than the first and last in the template must not appear as the first, or last, identifier in any template in the specification.

Standard Z imposes the rule that no two infix templates may be assigned the same precedence but different associativities. This rule is not imposed by ProofPower-Z, which instead prohibits use of the resulting syntactic abbreviations in a way which would be ambiguous.

We use the term *fancy-fix syntax* to refer to a use of a syntactic abbreviation.

6.4.5 Given Set Definition

A given set definition lists the names of the given sets being defined, optionally followed by a constraint:

BNF

```
│    GivenSet   =   ‘[‘, DecName, {‘,‘, DecName}, ‘]‘, [‘&‘, Constraint];
```

6.4.6 Abbreviation Definition

BNF

```
│    AbbDef     =   EqDef
│               |   SchemaBox;
│
│    EqDef      =   DefLhs, (‘≙‘ | ‘==‘), Schema;
│
│    SchemaBox  =   SCH, DefLhs, IS, VSchemaText, END;
│
│    DefLhs     =   ([VarName], {Id, VarName}, IdDec, [VarName]) − Id
│               |   VarName, [GenFormals];
│
│    GenFormals =   ‘[‘, DecName, {‘,‘, DecName}, ‘]‘;
```

An instance of the first alternative for *DefLhs* must match some template in some *gen* fixity paragraph in the specification, in the sense that the template results if we delete any decoration from the *DefLhs* and replace each *VarName* in it by ‘_’.

Note that the *Expr* possibility for *Schema* allows an abbreviation definition to be used either as a “horizontal” equivalent of the *SchemaBox* form or as a means of giving an equational definition of a, possibly generic, global variable.

6.4.7 FreeType Definition

In a free type definition, constructors which appear in a *fun* fixity paragraph may be written using fancy-fix syntax:

BNF

FreeTypeDef	=	*DecName*, ‘::=‘, *Branch*, {, ‘|‘, *Branch*}, [‘&‘, *FreeTypeDef*];
Branch	=	([*FreeTypePar*], {*Id*, *FreeTypePar*}, *IdDec*, [*FreeTypePar*])
	|	*VarName*, [*FreeTypePar*];
FreeTypePar	=	*Expr*;

We say that the first alternative for a branch matches a template if the template may be obtained from the branch by deleting any decoration and replacing each *FreeTypePar* by ‘...‘ or ‘_‘. Each branch construed under the first alternative for *Branch* must match some template in some *fun* fixity paragraph in the specification.

For compatibility with other dialects of Z, the expressions in a *FreeTypePar* may be enclosed in guillemet brackets as described in section 6.5.4.

6.4.8 Axiomatic Box

An axiomatic box has an optional list of generic formal parameters, a declaration, and an optional predicate.

BNF

AxBox	=	AX, [*GenFormals*, BAR], *VSchemaText*, END;

6.4.9 Constraint

A constraint may optionally start with a list of generic formal parameters.

BNF

Constraint	=	[*GenFormals*], *Pred*;

There is an ambiguity between this rule and the second alternative of the rule for the non-terminal *Expr3*, since a phrase of the form $[X]\,Y$ could be interpreted either as a constraint with generic formals $[X]$ or as a term beginning with an application of a horizontal schema expression to Y. The former interpretation is to be preferred (and, indeed, the second interpretation cannot be well-typed).

6.4.10 Conjecture

A conjecture comprises an optional label, an optional list of formal parameters and a predicate. A conjecture serves either for documentary purposes or as a means of recording an assertion for subsequent proof: the ProofPower system checks that it the conjecture is type-correct and then stores it in the theory database (whence it may be retrieved using the function *get_conjecture*).

BNF

Conjecture	=	[*Id*], ‘?⊢‘, [*GenFormals*], *Pred*;

As with *Constraint* there is an ambiguity between this rule and the second alternative of the rule for the non-terminal *Expr3*. The interpretation as a generic conjecture rather than as a phrase beginning with an application of a horizontal schema expression is to be preferred.

6.4.11 Declaration

A declaration is a sequence of basic declarations each of which is either an explicit declaration of a list of variables ranging over some set or a schema-as-declaration.

BNF

Decl	=	*BasicDecl*, {‘;‘, *BasicDecl*};
BasicDecl	=	*DecName*, {‘,‘, *DecName*}, ‘:‘, *Expr*
	|	*Schema*;

6.4.12 Schema

The syntax for schema-expressions is as follows:

BNF

Schema	=	*Schema2*
	|	*SQuant*, *SchemaText*, ‘$\bullet$‘, *Schema*;
SQuant	=	‘$\forall_s$‘ | ‘$\exists_s$‘ | ‘$\exists_{1\,s}$‘ | *Quant*;
Quant	=	‘$\forall$‘ | ‘$\exists$‘ | ‘$\exists_1$‘;
Schema2	=	*Schema3*
	|	*Schema2*, *SchInOp1*, *Schema2*;
	|	*Schema2*, *SchInOp2*, *Schema2*;
SchInOp1	=	‘$\wedge$‘ | ‘$\vee$‘ | ‘$\Rightarrow$‘ | ‘$\Leftrightarrow$‘ ;
SchInOp2	=	‘$\wedge_s$‘ | ‘$\vee_s$‘ | ‘$\Rightarrow_s$‘ | ‘$\Leftrightarrow_s$‘ | ‘$\upharpoonright_s$‘ | ‘$\fatsemi_s$‘;
Schema3	=	*Schema4*
	|	*SchPreOp*, *Schema3*;
SchPreOp	=	‘$\neg$‘ | ‘*pre*‘ | ‘Δ‘ | ‘Ξ‘;
Schema4	=	*Expr0*
	|	*Schema4*, [*RenameList*]
	|	‘[‘, *SchemaText*, ‘]‘
	|	*Schema4*, ‘$\backslash_s$‘, ‘(‘, *DecName*, {‘,‘, *DecName*}, ‘)‘;
RenameList	=	‘[‘, *Renaming*, {‘,‘ *Renaming*}, ‘]‘;
Renaming	=	*DecName*, ‘/‘, *DecName*;

The grammar for *Schema2* is ambiguous. The ambiguities are resolved by taking the operators of *SchInOp1* and *SchInOp2* as listed in decreasing order of precedence and taking each operator as right associative. The corresponding operators in *SchInOp1* and *SchInOp2* have equal precedence.

The operators of *SchInOp1* are overloaded: they are also used for the logical connectives. The rules for resolving the overloading are given in section 6.4.14 below. The operators of *SchInOp2* are not overloaded and can only denote schema operations.

6.4.13 Schema Text

The grammar distinguishes between vertical schema text and horizontal schema text to ensure that declaration and predicate parts are separated by a horizontal bar when arranged vertically and by a vertical bar when arranged horizontally.

BNF

```
VSchemaText     =     [Decl], [ST, Pred];

SchemaText      =     [Decl], [‘|‘, Pred];
```

6.4.14 Predicate

The syntax for predicate is as follows. Note that a schema-expression, and so an arbitrary expression, is one of the possibilities for a predicate.

BNF

```
Pred       =    Pred1, { ‘;‘, Pred1 }


Pred1      =    Pred2
           |    Quant, SchemaText, ‘•‘, Pred1
           |    ‘let‘, EqDef, {‘;‘ EqDef}, ‘•‘, Pred1;

Pred2      =    Pred3
           |    Pred2, LogInOp, Pred2;

LogInOp    =    ‘∧‘ | ‘∨‘ | ‘⇒‘ | ‘⇔‘;

Pred3      =    Pred4
           |    ‘¬‘, Pred3;

Pred4      =    Expr, Rel, Expr {Rel, Expr}               (* Pred4.A *)
           |    ([Expr], {Id, Exprs}, Id, [Expr])
                − (Id | (Expr, Id, Expr))                 (* Pred4.B *)
           |    Schema
           |    ‘true‘
           |    ‘false‘
           |    ‘(‘, Pred, ‘)‘;

Rel        =    Id | ‘∈‘ | ‘=‘;
```

The grammar for predicates is ambiguous. The ambiguities are resolved by imposing the following rules:

1. The operators in the production for *LogInOp* are listed in decreasing order of precedence, and are right associative.
2. In *Pred4.A* the *Id* in each *Rel* must appear in a template of the form _*Id*_ in a *rel* fixity paragraph.
3. The alternative *Pred4.B* is only allowed when the resulting phrase matches a template in a *rel* fixity paragraph in the specification. Here the phrase is said to match a template if the template can be obtained from it by replacing each direct constituent *Expr* in the phrase by '_' and replacing each direct constituent *Exprs* by '...'.
4. Negation is to have higher precedence than the infix schema operators of section 6.4.12
5. An argument corresponding to a stub of the form '_?' is to be construed as a *Pred*.
6. An argument corresponding to a stub of the form '_!' is to be construed as a *Pred* or a *Schema* according to whether the surrounding phrase is to be construed as a *Pred* or a *Schema*.
7. A phrase which can be construed both as a *Pred* and a *Schema* and not governed by rules 5 or 6 above should be construed as a *Pred*, unless this makes some enclosing phrase impossible to construe.

Rule 2 above controls the resolution of the overloading in Z of the logical connectives, $\forall$, $\exists$, $\exists_1$, $\land$, $\lor$, $\Rightarrow$, $\Leftrightarrow$and $\neg$, which are used both to construct schema expressions and to construct predicates. This resolution is done on the basis of syntactic context only, i.e. it can be done without a knowledge of the types of a construct and its constituents.

In some syntactic contexts where a construct formed with one of these connectives may appear, only a truth value, i.e. a predicate, is semantically acceptable, whereas, in other contexts either a set of bindings, i.e. a schema-as-expression, or a truth value would be acceptable, and in other contexts a truth value is not considered to be syntactically acceptable (because standard is a first order language).

The syntactic contexts demanding predicates are the positions where *Pred* occurs in *SchemaBox*, *AxBox*, *Constraint*, *Conjecture* and *SchemaText*; the syntactic contexts permitting either predicates or expressions are the operand positions of the logical connectives; and the contexts demanding expressions are the operand positions of the other schema operators, the *Pred* which occurs in *Term*, and all positions where *Expr*, *Expr0*, *Expr1*, *Expr2*, *Expr3* and *Expr4* appear. The rule is that the overloading of the logical connectives is to be resolved in favour of the predicate interpretation except where so doing would force a predicate to appear in some enclosing context which demands an expression.

As stated in rules 5 and 6 above, templates containing stubs of the form '_?' and '_!' provide exceptions to the general rule. See section 6.5.1 and 6.5.2 below for useful examples of such templates.

6.4.15 Expression

The syntax for expressions is as follows. Note that a predicate in brackets is one of the alternatives for expressions. This is because extended Z is higher-order not first-order. In standard Z, the predicate must take the *Schema* alternative of *Pred4* unless the expression appears, perhaps nested in additional (and redundant) layers of brackets, as an immediate constituent of one of the alternatives for *Pred*, *Pred21* or *Pred3*.

BNF

```
Expr        =  Expr1;

Expr0       =  Expr1
            |  ‘μ‘, SchemaText, ‘•‘, Expr0
            |  ‘λ‘, SchemaText, ‘•‘, Expr0;
            |  ‘let‘, EqDef, {‘;‘ EqDef}, ‘•‘, Expr0;

Expr1       =  Expr2
            |  ([Expr1], {Id, Exprs}, Id, [Expr1]) − Id;   (∗ Expr1.A ∗)
            |  Expr1, ‘×‘, Expr1, {‘×‘, Expr1};           (∗ Expr1.B ∗)

Expr2       =  Expr3
            |  ‘ℙ‘, Expr2;

Exprs       =  [Expr, {‘,‘, Expr}];

Expr3       =  Expr4
            |  Expr3, Expr4
            |  ‘θ‘, Expr4, [Decor];

Expr4       =  VarName, [GenActuals]
            |  Literal
            |  Quotation
            |  ‘(‘, Pred, ‘)‘
            |  ‘(‘, Schema, ‘)‘, [Decor]
            |  ‘(‘, ‘μ‘, SchemaText, ‘)‘
            |  ‘(‘, Expr, {‘,‘, Expr}, ‘)‘
            |  Schema
            |  ‘(‘, [EqDef, {‘,‘ EqDef}], ‘)‘
            |  ‘⟨‘, [Expr, {‘,‘, Expr}], ‘⟩‘
            |  ‘{‘, [Expr, {‘,‘, Expr}], ‘}‘                    (∗ Expr4.A ∗)
            |  ‘{‘, SchemaText, [‘•‘, Expr], ‘}‘                (∗ Expr4.B ∗)
            |  Expr4, ‘.‘, VarName
            |  Expr4, ‘.‘, Natural                        (∗ Expr4.C ∗)
            |  Quotation;

GenActuals  = ‘[‘, Expr, {‘,‘, Expr}, ‘]‘;

Literal     =  Natural | Float | Character | String;
```

The third alternative for *Expr4* here is part of extended Z and is not allowed in standard Z.

The grammar for expressions is ambiguous, the ambiguities are to be resolved using the following rules.

1. The alternative *Expr1.A* is only allowed when the resulting phrase matches a template in a *gen* or *fun* fixity paragraph in the specification. Here the phrase is said to match a template if the template can be obtained from it by replacing each direct constituent *Expr* in the phrase by ‘_’ and replacing each direct constituent *Exprs* by ‘...’. (In extended Z, *Expr1.A* is also allowed when the phrase matches a template in a *rel* fixity paragraph.)
2. The alternative *Expr1.A* corresponding to a *gen* fixity paragraph has lower precedence that the alternative *Expr1.B* which has lower precedence than the alternative *Expr1.A* corresponding to a *fun* fixity paragraph. The precedence and associativity in the fixity paragraphs give the precedence and associativity to apply for a phrase which can be interpreted in two ways using *Expr1.A* with the same sort of fixity paragraph. (In extended Z, a *rel* fixity paragraph is treated as right associative with precedence *0*). A phase which can be interpreted in two ways using *Expr1.A* and for which the relevant precedences are different but the relevant associativities are the same is not allowed.
3. A phrase of the form $\{S\}$ where S is a *Name* could be interpreted as a set display (*Expr4.A*) or as a set comprehension (*Expr4.B*). The set display is to be preferred.
4. A phrase of the form $V[S]$ where V is a *Name* could be interpreted as a generic instantiation (*Expr4.A*) or as application of V to a horizontal schema (*Schema4.A*). The generic instantiation is to be preferred.
5. A phrase which can be construed both as a *Schema* and an *Exp* should be construed as an *Exp*, wherever possible (this is not a source of semantic ambiguity).
6. In extended Z, the ambiguity between the fourth and fifth alternatives of *Expr4* is to be resolved in favour of the former (there is no semantic ambiguity here).
7. A token should not be treated as a floating point literal if it is preceded by the reserved identifier ‘.‘’. I.e., the alternative *Expr4.C* above is to be preferred over the production for *Float* in section 6.3.

6.4.16 Names

BNF

VarName	=	*IdDec*
	|	‘(‘, (([*Stub1*], {*Id*, *Stub*}, *IdDec*, [*Stub1*]) − *IdDec*), ‘)‘;

BNF

DecName	=	*IdDec*
	|	(([*Stub1*], {*Id*, *Stub*}, *IdDec*, [*Stub1*]) − *IdDec*)
	|	*Quotation*;

In the first alternative in both of the above productions the *Id* in the *IdDec* must not appear in any fixity paragraph in the specification. In the second alternative the result of deleting any decoration must make the phrase between the brackets the same as some template in some fixity paragraph in the specification.

In the third alternative for *DecName*, the nested quotation must be an HOL variable or constant.

BNF

IdDec	=	*Identifier*, [*Decor*];

6.5 LANGUAGE SUPPORT LIBRARY

Certain features, such as conditional predicates and expressions, that are language constructs in other dialects of Z can be defined in ProofPower-Z. Also, in order to enter fragments of Z while doing proofs, some additional language features, such as casts, are desirable and these too can be defined in the language. This section describes the library that provide these features.

6.5.1 Casts

The global variables '$\_ \overset{\oplus}{\oplus} \_$' and '$\Pi\ \_$' are defined in the library of theories which support ProofPower-Z as if by the following paragraphs; however, they are treated specially by the parser and type checker.

Informal Z

$\mathit{fun}\quad \_ \overset{\oplus}{\oplus} \_$

Z

$[X]$

$\_ \overset{\oplus}{\oplus} \_ : X \times \mathbb{P}\, X \to X$

$(\_ \overset{\oplus}{\oplus} \_) = \mathit{first}$

Informal Z

$\mathit{fun}\quad \Pi\ \_?$

Z

$[X]$

$\Pi\ \_? : \mathbb{B} \to \mathbb{B}$

$\forall x \bullet \Pi\ x = x$

'$\_ \overset{\oplus}{\oplus} \_$' is an operator which acts as a type cast, i.e., it ensures that its left operand has the same type as the elements of its second operand and it takes the same value as its left operand. This is intended for use during proof when it is necessary to enter Z expressions which do not contain enough information to fix the types of all their sub-expressions. For this reason, an expression of the form $ⓩx \overset{\oplus}{\oplus} s⌝$ is treated as identical with $ⓩx⌝$, except that during type-checking the type constraints imposed by the above definition of '$\_ \overset{\oplus}{\oplus} \_$' are enforced.

In entering Z terms, one occasionally needs to provide a syntactic context which will force the schema interpretation of the logical connectives. The left-hand operand of '$\_ \overset{\oplus}{\oplus} \_$' supplies such a context. For example, $ⓩ[x{:}\mathbb{Z}] \land [x{:}\mathbb{Z}]⌝$ is a predicate conjunction, whereas $ⓩ([x{:}\mathbb{Z}] \land [x{:}\mathbb{Z}]) \overset{\oplus}{\oplus} U⌝$ is a schema conjunction.

'$\Pi\ \_$', while it acts as a prefix function from the point of view of the grammar for Z (i.e., it may be used to form expressions taking the alternative of *Expr0* above labelled *Expr1.AS*), supplies a syntactic context forcing the predicate interpretation of the logical connectives, and forcing the type checker to treat a schema-expression as a schema-as-declaration. For example, in $ⓩ[x{:}\mathbb{Z}] \land [x{:}\mathbb{Z}] \neq [x{:}\mathbb{N}]⌝$, the conjunction is necessarily a schema conjunction by the rules of section 6.4.14. One can force the interpretation as a logical conjunction and have the right hand-side treated as a schema-as-predicate by writing $ⓩ\Pi([x{:}\mathbb{Z}] \land [x{:}\mathbb{Z}]) \neq \Pi([x{:}\mathbb{N}])⌝$. Apart from this special interpretation during parsing and type-checking, $ⓩ\Pi\ x⌝$ is treated as identical with $ⓩx⌝$.

If for some reason, it is necessary to construct a Z term which actually has '$\_ \overset{\oplus}{\oplus} \_$' or '$\Pi\ \_$' as a sub-term, this may be done by not using the fancy-fix notation, for example, while $ⓩ1 \overset{\oplus}{\oplus} \mathbb{Z}⌝$ is just the same as $ⓩ1⌝$, $ⓩ(\_\overset{\oplus}{\oplus}\_)(1,\ \mathbb{Z})⌝$ really is an application of $ⓩ(\_\overset{\oplus}{\oplus}\_)⌝$ to the pair $ⓩ(1,\mathbb{Z})⌝$.

6.5.2 Conditionals

Standard Z supports a conditional expression form which is defined as a higher-order operator in ProofPower-Z: the global variable '*if _? then _! else _!*' is defined in the library of theories which support ProofPower-Z as follows:

```
z
│function 0 if _? then _! else _!
```

```
z
╒═[X]══════════════════════════════
│ if _? then _! else _! : 𝔹 × X × X → X
├──────────────────────
│ ∀x, y:X•
│         (if true then x else y) = x
│ ∧       (if false then x else y) = y
└──────────────────────────────────
```

6.5.3 Guillemet Brackets

ProofPower-Z supports the syntax for free types in Standard Z by providing an outfix operator $\ll \_ \gg$. This makes the symbols "$\ll$" and "$\gg$" act as general purpose brackets that will not be eliminated by the type-checker. The guillemet brackets are defined as follows:

```
z
│function 0 ≪ _! ≫
```

```
z
╒═[X]══════════════════════════════
│ ≪ _! ≫ : X → X
├──────────────────────
│ ∀x:X• ≪x≫ = x
└──────────────────────────────────
```

6.5.4 Underlined Infix Relations

An arbitrary expression can be used as an infix relation symbol in ProofPower-Z by enclosing it in the underlining brackets, $\underline{(}$ and $\underline{)}$. These cause the expression to be underlined when printed. The underlining brackets are defined as follows:

z

$|\,\mathit{relation}\ \_\ \underline{\ \_\ }\ \_$

z

$[X, Y]$

$\_\ \underline{\ \_\ }\ \_ : \mathbb{P}(X \times \mathbb{P}(X \times Y) \times Y)$

$\forall x{:}X;\ R : \mathbb{P}(X \times Y);\ y{:}\ Y \bullet x\ \underline{\ R\ }\ y \Leftrightarrow (x,\ y) \in R$

REFERENCES

[1] Michael J.C. Gordon and Tom F. Melham, editors. *Introduction to HOL.* Cambridge University Press, 1993.

[2] Michael J.C. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF. Lecture Notes in Computer Science. Vol. 78.* Springer-Verlag, 1979.

[3] Leslie Lamport. *A Document Preparation System LaTeX, user's guide & reference manual.* Addison-Wesley, 1986.

[4] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[5] BS6154:1981. *Method of defining syntactic metalanguage.* British Standards Institution, 1981.

[6] DS/FMU/IED/USR001. *ProofPower Document Preparation.* Lemma 1 Ltd.

[7] DS/FMU/IED/USR004. *ProofPower Tutorial Manual.* Lemma 1 Ltd., `http://www.lemma-one.com`.

[8] DS/FMU/IED/USR007. *ProofPower Installation and Operation.* Lemma 1 Ltd., `http://www.lemma-one.com`.

[9] DS/FMU/IED/USR011. *ProofPower Z Tutorial.* Lemma 1 Ltd., `http://www.lemma-one.com`.

[10] DS/FMU/IED/USR013. *ProofPower HOL Tutorial Notes.* Lemma 1 Ltd., `http://www.lemma-one.com`.

[11] LEMMA1/HOL/USR029. *ProofPower HOL Reference Manual.* Lemma 1 Ltd., `rda@lemma-one.com`.

INDEX