Programming Research Group

A CONCRETE Z GRAMMAR

Peter T. Breuer
Jonathan P. Bowen

PRG-TR-22-95

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

A Concrete Z Grammar*

Peter T. Breuer †
Jonathan P. Bowen ‡

Abstract

This report presents a concrete grammar for the formal software specification language Z, based on the BNF-like syntax description in the widely used Z Reference Manual. It has been used as a starting point for several projects associated with Z. The grammar is written in the format required for the public domain compiler-compiler PRECC. It has also been used as a basis for grammars aimed at other compiler-compilers, including *yacc* and PCCTS. An important goal in publishing it here is to make a working concrete grammar for Z publicly available and thus to promote the production of Z-based utilities. Another intention of this report is to report on the use of what is itself a high-level formal specification language with a formally defined parser semantics: PRECC. It is used here to define a human-readable language, namely Z, that has an ambiguous and context-sensitive syntax, and we also report on engineering aspects of the work.

---

*Further copies of this Technical Report may be obtained from the Librarian, Oxford University Computing Laboratory, Programming Research Group, Wolfson Building, Parks Road, Oxford OX1 3QD, England (Telephone: +44-865-273837, Email: `library@comlab.ox.ac.uk`).

†Departamento de Ingeniería de Sistemas Telemáticos, Universidad Politécnica de Madrid, ETSI Telecomunicación, Ciudad Universitaria, E-28040 Madrid, Spain (Email: `ptb@dit.upm.es`).

‡Address from 1st October 1995: The University of Reading, Department of Computer Science, Whiteknights, Reading, Berks RG6 6AY, England (Email: `J.P.Bowen@reading.ac.uk`).

1 Introduction

The syntax summary for Z given in Chapter 6 of the widely used Z Reference Manual [14] and a similar but more concrete description in Section 8 of the *f*UZZ type-checker manual [13] (henceforth these will both be referred to as the 'ZRM' syntax or grammar) provide a useful basis for the front-end parsers in tools aimed at this popular software specification language. The grammars are expressed in a relatively pure BNF style, which makes them readily adaptable to the particular specification languages of standard parser generators such as *yacc* [9]. The problem is, however, that, as they stand, they require considerable adaptation before *yacc* or more modern generators can use them. Even then, variations in semantics or the limitations of different parser generators may eventually make the relationship of the generated application software to the intended grammar less than immediately obvious. Taking account of the effort required to adapt and debug any concrete grammar, and the multitude of existing projects known to us, both commercial and academic, which are doing this because they need a working Z parser as a starting point, it has seemed worthwhile putting effort into a concrete public domain grammar that can be used as the basis for further projects. Several groups and individuals have suborned the first versions of this grammar for their projects, and have collaborated in its testing and refinement. We have taken the engineering approach of using a parser generator that can use the published BNF (almost) as it stands, and of relying on the compositionality embedded in the generator to do most of our thinking for us. See [4] for a justification of this approach and a comparative study. The idea is that if we can get each part of the grammar right, then the whole will be right, or at least, it will conform to the published specification.

One aim of this document is thus to set out a working grammar – in as brief a space as possible. The grammar is in the form of a script for the PRECC utility [3, 4] – a publicly available compiler-compiler which accepts scripts written in an extended BNF and allows the use of context sensitive and attribute grammars [7]. The practical effect is that the ZRM grammar can be used almost as it stands. There is about an 80% carry-over. Most top-down parser generators will do a fair job of allowing the form of the grammar to be preserved; PRECC makes this goal easier to achieve than most. A second aim of this document is to test the capacity of PRECC to cope with a human-readable language with an occasionally ambiguous and generally context sensitive syntax. A third aim is to report on the engineering aspects of this kind of specification-oriented approach to building a parser.

There are several technical reasons why one should expect that PRECC would handle a Z grammar much more easily than *yacc*. In the first place, it is generally unnecessary to decorate the production rules with the sort of attribute ('semantic') information that might be required in order to disambiguate the parse. Generated parsers have unbounded lookahead – where *yacc* parsers have only one-token lookahead – so problems over disambiguation are not so severe. As top-down parsers, PRECC parsers intrinsically 'know' how they arrived at a particular point and may then read the same concrete symbols in different ways according to that context. This is entirely natural for human beings too and conforms to the way the ZRM syntax specification has been written. It assumes, for example, that the same symbols on the page may have to be read as identifiers or as schema names according to the context in which they are found. But this kind of inherent disambiguation is difficult to achieve with *yacc*-based parsers, often requiring modification of the basic paradigm. *Yacc* parsers are bottom-up and normally have no contextual memory and extra state-based context may have to be explicitly added to the code. Although the determined programmer may do so, the result may be a parser that works but has no obviously verifiable reason for doing so.

Certainly, a stateless (i.e. pure) *yacc* grammar for Z cannot be written because several parts of the ZRM grammar call for different productions from the same symbol sequences. The productions labelled *Gen_Formals* and *Gen_Actuals*, for example, both clearly match token input of the shape '`( foo, bar, gum )`' (in which `foo`, `bar`, etc. are tokens labelled as *L_WORD*s by the lexer). A *yacc* parser in the middle of such a sequence may not know which reduction to make after the final closing parenthesis (to a *Gen_Formals* or a *Gen_Actuals*). Only context information can disambiguate the parse. The situation in particular *yacc* grammars may vary – the ZRM grammar has to be approximated in a *yacc* implementation. It may be that a particular *yacc* grammar restricts the circumstances in which *Gen_Formals* and *Gen_Actuals* are sought with the result that there is no conflict between these two. We have received one report of a *yacc* script that avoids problems here. On the other hand, at least one other correspondent reports that a *yacc* grammar following the ZRM *does* generate conflicts at this point.

Secondly, it seems that it may be difficult to write an independent lexer for Z, because of the way that schema names and other identifiers have to be distinguished during the parse. Certainly there is no lexical basis for the distinction, so feedback between parser and lexer seems to be required. Typically, lexers are written to expect a fair bit of communication from the parser (and vice versa, of course), but PRECC can serve as lexer as well as parser, and then the inherent context-sensitivity of PRECC disambiguates the lexical as well as the parsing stages without the need for explicit communication. The generated lexer will be looking for different interpretations of the same characters in different parse contexts, but the specification script will still look entirely natural. Somewhat the same advantage accrues to all two-level parser generators. For example, PCCTS [11], which has a finite lookahead parser, generates a lexer from the grammar script and allows attributes on *lexemes* (the atomic lexical units) which may be used as information by the parser.

Furthermore, the full semantics of PRECC is published [3], along with a logic for reasoning about grammars and parses. Briefly, PRECC has a trace/refusal semantics similar to CSP [8], and admits the same kind of formal verification techniques with respect to specification and the parser as does, for example, Occam with respect to CSP specifications and machine code compiled from Occam programs. So at least some kind of formal testing and verification is possible, apart from the usual kind of testing, and this is appropriate in the formal methods setting in which Z is situated.

What of the efficiency to be expected of the result? The study part of [4] shows comparable runtime efficiencies to *yacc* (in fact, approximately 30% worse in the case study, but the times were still fractions of a second for the test scripts, and the grammar studied was derived from a *yacc* grammar original with *yacc* orientations, so the results are skewed towards *yacc*). Memory resources might be a problem in a top-down parser where they cannot be a problem in the automaton built by *yacc*, but normally the memory usage is comparable. A top-down parser executing recursions to a depth of a thousand calls, for example, will use about thirty to forty thousand bytes of stack space, which is not much in todays terms. It is difficult to conceive of a Z schema which requires a depth of more than one hundred stacked calls for its analysis at any point, but, because there is no limit on the complexity of a Z schema, equally, there may be no limit on the resources required for its analysis. Simply putting one hundred pairs of parentheses around an expression will stack one hundred calls before the expression can be resolved. It is not even the case that the resources required necessarily scale linearly. That depends on the way the grammar is written.

Those tests that we have conducted show no resource tie-ups, but an extremely careful complexity analysis would be required to prove that none occurs. It is only necessary to check that there are no production rule circuits between terminals in the grammar in order to guarantee linear maximum call depth behaviour, but the constant involved needs to be small. Time-wise complexity, while also amenable to static analysis, depends intrinsically on the numbers of alternate productions. In the worst case, a test script may elicit the maximum number of failed alternate parses at every possible branch point. The grammar has to be written so that in practice, failing alternates are weeded out early. The extreme example of such is a grammar written for *yacc* itself, in which every alternate production must be distinguished by the next incoming lexeme!

What of the difficulties in designing and debugging a grammar for PRECC, as opposed to *yacc*? That question is largely answered in the detail of the text of this document. But a word at a more general level is appropriate. It is our experience that the only suitable starting point for a grammar script is a BNF description. Trying to construct a grammar from practical experience of the language alone is very difficult, more so if that experience is less than comprehensive. In general, there are two fields of knowledge that have to be married here:

- target language (Z) syntax and semantics;
- parser description (PRECC/*yacc*) syntax and semantics.

An expert from either side of the domain can cross part-way over using one of the two specification intermediates:

- a standard syntax interchange format (BNF);
- a parser description script (PRECC/*yacc*).

But it may be unusual for one expert to be equally familiar with both sides. In the case of *yacc*, constructing a parser description script requires a great deal of understanding of the particular semantics of *yacc* parsers. The task is easier for a language expert constructing a PRECC parser because of the compositional nature of scripts for the latter.

We have approached the problem from the side of the divide that entails greater familiarity with the parser semantics than the target language, however. We have taken a Z language description that has been constructed by an independent expert, and endeavoured to follow it without question. The ambiguities that we encountered were omissions rather than bugs, and required additions to the published grammar rather than corrections. As a result, we feel that we have succeeded in correctly rendering the published description.

2 Background

Z was originally designed as a formal specification notation to be read by humans rather than computers and, as such, the notation remained relatively fluid for many years during its early development [5]. In this period there were few tools available. The main support was for word processing, providing the special mathematical fonts for Z symbols and the facility to produce Z schemas. This did not even provide or enforce syntax checking of the specifications. Over the years, the syntax [14] and semantics [12] of Z have become more settled, although the current international ISO standardization effort [6] still allows room for debate.

A number of useful Z tools are now available. For example *f*UZZ [13] and ZTC [15] provide type-checking facilities. These must parse a concrete representation of Z. Currently various Z style files for the LaTeX document preparation system [10] are widely used to prepare Z specifications which can subsequently be machine-checked using such tools. The proposed Z standard [6] includes an interchange format based on SGML which is likely to become widely used by tool developers once the Z standard is accepted, but this is still some way off.

Parsing the concrete representation of a Z specification is an important part of any tools used to process the specification. One simple approach is to use a representation that is easily handled directly by a particular tool (e.g., see theorem proving support for Z using HOL in [1]). This is fine for prototype tools, but a production tool will need to handle whatever the standard concrete representation is deemed to be, be it LaTeX source format or the Z standard interchange format, for example. As remarked, publicly available machine-readable Z syntax descriptions that are useful for tool builders are currently thin on the ground and this report is intended to help correct the omission.

3 The Grammar Script

As in the ZRM grammar, the entry point for a parser is a *Specification*, which consists of a sequence of *Paragraph*s. The ZRM specifies that the sequence must be non-empty, but it is possible to permit the empty sequence too (it has the empty semantics).

```
@ Specification = Paragraph*

MAIN(Specification)
```

We augment the ZRM specification of paragraphs by including two more alternatives, namely *Directive* and *Comment*. A *Directive* is a mode change instruction, such as a declaration of binding power for an infix operator. A *Comment* is to stand for anything that is not parsed as Z. This seems more natural than making non-Z vanish through being treated as 'white space' by the lexer, which would then have to be fundamentally two-state. Note that the semantics here is order-sensitive. In case of ambiguity, first the text will be tried as a match to an *Unboxed_Para*, then as an *Axiomatic_Box*, etc., and finally it will be tried as a *Comment*.

```
@ Paragraph     = Unboxed_Para
@               | Axiomatic_Box
@               | Schema_Box
@               | Generic_Box
@               | Directive
@               | Comment
```

A directive declares that a symbol is of a particular lexical kind. When a LaTeX text is parsed, for example, the "`%%inop`" directive will have the side-effect of augmenting the list of symbols detected as infix in subsequent paragraphs. A predefined set of infix symbols is generally set up in a Z prelude script.

Because a directive can contain only a restricted subset of the standard white space characters, a final *L_ENDLINE* is specified here. It explicitly matches the end of line that cannot appear within the directive itself. If a single level grammar were being defined, it would be a dummy actuating a mode change in the lexer in order to allow line breaks as white space again. Here it really is intended to match a symbol passed up from the lower level of the grammar.

```
@ Directive     = L_PERCENT_PERCENT_INOP   Symbols Priority\n L_ENDLINE
@                   {: add_inops($n); :} !          /* e.g. %%inop * \div  4   */
@               | L_PERCENT_PERCENT_POSTOP Symbols L_ENDLINE
@                   {: add_postops(); :} !          /* e.g. %%postop \plus     */
@               | L_PERCENT_PERCENT_INREL  Symbols L_ENDLINE
@                   {: add_inrels(); :}  !          /* e.g. %%inrel \prefix    */
```

```
@               | L_PERCENT_PERCENT_PREREL Symbols L_ENDLINE
@                  {: add_prerels(); :} !         /* e.g. %%prerel \disjoint*/
@               | L_PERCENT_PERCENT_INGEN  Symbols L_ENDLINE
@                  {: add_ingens(); :}  !         /* e.g. %%ingen \rel      */
@               | L_PERCENT_PERCENT_PREGEN Symbols L_ENDLINE
@                  {: add_pregens(); :} !         /* e.g. %%pregen \power_1 */
@               | L_PERCENT_PERCENT_IGNORE Symbols L_ENDLINE
@                  {: add_whitesp(); :} !         /* e.g. %%ignore \quad    */
@               | L_PERCENT_PERCENT_TYPE   Symbols L_ENDLINE
@               | L_PERCENT_PERCENT_TAME   Symbols L_ENDLINE
@               | L_PERCENT_PERCENT_UNCHECKED      L_ENDLINE
@                                                 /* e.g. %%unchecked       */
```

The "`%%unchecked`" directs that the next paragraph not be type-checked. Most parsing errors should then also be ignored, but we will ignore that! The parser should switch to a more forgiving mode but it is too much trouble here to define a new grammar just for that. The "`%%type`" and "`%%tame`" directives can also be ignored from the point of view of parsing alone.

Why is the above set of definitions so complex? Getting the PRECC parser to dynamically update the table of infix operators, for example, requires the extra annotations here. They deal with *attributes* attached to the parsed terms. Attributes are declared as "*n*" and dereferenced as "*$n*". They may be passed into *action*s, which are pieces of C code enclosed between a "`{:`" ... "`:}`" pair. Pending actions are discharged when the parse passes through a point denoted with an exclamation mark in the grammar. This is the only point in the script where actions will be introduced, as they are used only to make the required parser mode changes. In this case the action is a side-effecting update of a global table via the call "`add_inops($n)`", and it is executed immediately because the exclamation mark follows it in the definition.

An auxiliary action during the parsing of *Symbols* loads a buffer with an array of integer keys corresponding to the symbols just seen. Each symbol carries a unique key as an attached attribute. The key is generated by the lexer. The calling *Directive* now saves the information and later it will be examined by other parts of the parser.

```
int tsymcount=0; int tsymbuff[MAXSYMS];

@ Symbols       = {: tsymcount=0; :}
@                 Symbol*                         /* allow zero or more     */

@ Symbol        = L_WORD\x                        /* e.g. \quad or + or foo */
@                 {: tsymbuff[tsymcount++] = $x; :}

@ Priority      = L_PRIORITY                      /* single digit in [1-6]  */
```

We will allow a comment to consist of any positive sequence of symbols at all, so the top level parser certainly cannot fail. Here *L_COMMENTCHAR* is the label supplied by the lexer, and, because the lexer will be context sensitive – i.e., it will only look for a *L_COMMENTCHAR* when it is told to look for one by the parser – so it will be safe to allow almost any character to match here.

```
@ Comment       = L_COMMENTCHAR+
```

In constructing this grammar, we have endeavoured to stay as close to the structure of the *f*UZZ description as possible. The reason is that any differences carry with them the possibility of introducing an overt error, independent of the covert errors that may accrue through mistakes in the implementation or design of the parser generator. And wherever we have had to differ, special attention has been paid to correctness of the revised form with respect to (the BNF semantics of) the original in order to avoid mistakes that may derive from a third form of error – miscomprehension of specification language semantics. But here we have been able to follow the *f*UZZ description closely, even to layout details. We can thus be confident that this part of our work is correct.

Next we define the detailed forms of the several varieties of unboxed paragraph, and then the three forms of boxed paragraphs. There is no problem in distinguishing these from each other because they each begin with a concretely different lexeme.

```
@ Unboxed_Para  = L_BEGIN_ZED                      /* \begin{zed}              */
@                       Item {Sep Item}*           /* foo \also bar \\ more    */
@                     L_END_ZED                    /* \begin{zed}              */
```

Inside any unboxed paragraph several individually large items of text may appear, separated by the legal separators. The separators will be defined formally below, but for the record, concretely, they are the semicolon, the latex line separator "`\\`" and the Z separator "`\also`".

There are five sorts of these items. It is always a good idea to place at the front of a list of parsing alternatives those clauses that do have distinguishing features, so that they can be eliminated from consideration by the parser early on. Here, the sequence of identifiers within a bracket pair is distinguished by the leading open bracket (not a parenthesis) and should be listed first. Of the next three clauses, each contains a distinguishing token, respectively *L_DEFS*, *L_EQUALS_EQUALS* and *L_COLON_COLON_EQUALS*. The order of the three displayed here is actually that given in the *f*UZZ grammar, but it would be preferable on general principles to invert the order and place the clause with *L_COLON_COLON_EQUALS* at the head, because that token definitely must occur in second place while the other two might occasionally occur later.

```
@ Item          = L_OPENBRACKET Ident {L_COMMA Ident}* L_CLOSEBRACKET
@                                                  /* [ foo,fie,fee,fum ]          */
@               | Schema_Name [ Gen_Formals ] L_DEFS Schema_Exp
@                                                  /* foo [ fie,fee ] \defs fum    */
@               | Def_Lhs L_EQUALS_EQUALS Expression
@                                                  /* foo [ fie,fee ] == fum       */
@               | Ident L_COLON_COLON_EQUALS Branch {L_VERT Branch}*
@                                                  /* foo ::= fie | fee | fum      */
@               | Predicate
```

A *yacc* parser may be able to distinguish these clauses, but the second and the third seem particularly problematic. Can an *Ident* (which may start a *Def_Lhs*) be distinguished from a *Schema_Name*? Both are represented by *L_WORD* tokens from the tokeniser, so it would seem not. A *yacc* parser would have to wait until the following *L_DEFS* or *L_EQUALS_EQUALS* (or *L_COLON_COLON_EQUALS*) to decide what the interpretation of the tokens it has seen until then should have been, and that is sometimes too late for its one token look-ahead. There is no problem for PRECC because the parser will backtrack when necessary. So long as there is some token unique to a clause (relative to its alternate clauses), the clause will be distinguished correctly by a PRECC parser, no matter when the token comes.

We place a *Predicate* as the case of last resort. It will be checked for when all other alternatives have been rejected. A predicate might be only a single identifier.

The three kinds of boxed paragraph are distinguished by the leading lexeme, so present no problems.

```
@ Axiomatic_Box = L_BEGIN_AXDEF                       /* \begin{axdef}          */
@                     Decl_Part                       /*      foo               */
@               [ L_WHERE                             /* \where                 */
@                     Axiom_Part ]                    /*      bar               */
@                 L_END_AXDEF                         /* \end{axdef}            */

@ Schema_Box    = L_BEGIN_SCHEMA L_OPENBRACE Schema_Name L_CLOSEBRACE [Gen_Formals]
@                                                     /* \begin{schema}{fie}[fee]   */
@                     Decl_Part                       /*      foo               */
@               [ L_WHERE                             /* \where                 */
@                     Axiom_Part ]                    /*      bar               */
@                 L_END_SCHEMA                        /* \end{schema}           */

@ Generic_Box   = L_BEGIN_GENDEF [Gen_Formals]        /* \begin{gendef}[ fie,fee ]  */
@                     Decl_Part                       /*      foo               */
@               [ L_WHERE                             /* \where                 */
@                     Axiom_Part ]                    /*      bar               */
@                 L_END_GENDEF                        /* \end{gendef}           */
```

Descending to a little more detail here, and still following the *f*UZZ grammar, a boxed paragraph for a *gendef* contains a sequence in its top part and a sequence in its bottom part. The separator for the list will be the first symbol that cannot possibly be part of a *Basic_Decl* or *Predicate*, respectively. In other words, if a separator were valid within a predicate, then it would be parsed as part of the predicate. Only when the predicate part of the token stream has been exhausted is the separator searched for.

```
@ Decl_Part     = Basic_Decl {Sep Basic_Decl}*        /* foo \also bar          */

@ Axiom_Part    = Predicate {Sep Predicate}*          /* foo \also bar          */
```

Finally, we come to the definition of a separator promised above.

```
@ Sep           = L_SEMICOLON | L_BACKSLASH_BACKSLASH | L_ALSO
```

Note that the grammar is being expressed in terms of abstract lexemes received from the lexer, although the names have been chosen to reflect their usual concrete representations. They are usually LaTeX-compatible symbols. But they do not have to be. The only requirement is that differently named lexemes have different concrete representations. The lexer could, for example, be directed to recognize plain ASCII combinations instead.

The ZRM specification might fail at this point for a 1-token lookahead parser such as *yacc* working in combination with a separate lexer, because *Var_Name*, *Pre_Gen* and *Ident* are all *L_WORD* tokens supplied by the lexer. Some lookahead or context-sensitive disambiguation is necessary in contexts in which this production might arise.

We use a new term "*Pre_Gen_Decor*" to replace the pair "*Pre_Gen Decoration*" that appears in the *f*UZZ grammar. The decoration may be empty and the *Pre_Gen* never appears separately, so a single handle is convenient for us. Similarly for *In_Fun*, *Post_Fun*, *Post_Gen*, *In_Rel*, *Pre_Rel*.

```
@ Def_Lhs       = Var_Name [ Gen_Formals ]            /* foo [ fie,fee ]        */
@               | Pre_Gen_Decor Ident                 /* fie' foo               */
@               | Ident In_Gen_Decor Ident            /* foo fie' fum           */
```

The next production has had the order of alternates changed round from that which appears in the *f*UZZ specification. This is a question of parser semantics. Putting the longer parse first means that PRECC will try the long parse, and then, if it fails, try the short parse. The other way round, the short parse would be accepted where the longer one might have been a 'better' match.

```
@ Branch        = Var_Name L_LDATA Expression L_RDATA
@                                                   /* foo \ldata bar \rdata    */
@               | Ident                             /* foo                      */
```

The next production is in a very inefficient form for PRECC, but the published BNF has been followed as strictly as possible here. This would have been rendered more efficiently as a number of alternate opening symbols followed by the same continuation pattern in each case.

```
@ Schema_Exp    = L_FORALL Schema_Text L_AT Schema_Exp
@                                                   /* \forall foo : fie @ bar    */
@               | L_EXISTS Schema_Text L_AT Schema_Exp
@                                                   /* \exists foo : fie @ bar    */
@               | L_EXISTS_1 Schema_Text L_AT Schema_Exp
@                                                   /* \exists_1 foo : fie @ bar  */
@               | Schema_Exp_1                      /* bar                        */
```

The *f*UZZ script now distinguishes right and left associative operations, but this has little bearing on the parse itself, only on the way a parse tree might be built during the parse. It is important information, but it does not affect the correctness or otherwise of, for example, a schema expression, just the way it is interpreted. Still, the distinction has been preserved here, although it is not required for our purposes.

Right associative operations have been written using recursive productions. It is easier to adapt this kind of presentation to later build attributes in right-associative order, should that eventually be required. Here they are at the top level, including recursion. There is only one most weakly binding right associative operator.

```
@ Schema_Exp_1  = Schema_Exp_2 [ L_IMPLIES Schema_Exp_1 ]
@                                                   /* foo \implies bar           */
```

Now, one level down from the top, we search for a series of '*L_HIDE* parts' separated by left associative operations. There are a host of these most weakly binding left associative operators.

```
@ Schema_Exp_2  = Schema_Exp_3
@                 {  L_LAND | L_LOR | L_IFF | L_PROJECT | L_SEMI | L_PIPE
@                   Schema_Exp_3
@                 }*                                /* foo \land fie \lor fee     */
```

The '*L_HIDE*' part captures the distinctive trailing *L_HIDE* with its bits and pieces:

```
@ Schema_Exp_3  = Schema_Exp_U                      /* foo                        */
@                 { L_HIDE L_OPENPAREN Decl_Name {L_COMMA Decl_Name}* L_CLOSEPAREN }*
@                                                   /* \hide( fie,fee,fum )       */
```

The inherent ambiguity here is resolved deterministically by PRECC in favour of a longest possible initial sequence of consecutive *Schema_Exp_U* in a *Schema_Exp_3*. That is the sequence until the first *L_HIDE*, if there is one, and the whole sequence if there is not.

The components of the expression are now atomic units. They are *Schema_Exp_Us*. These are either bracket or parenthesis constructions containing higher level definitions, or plain *Schema_Refs*, possibly with tightly binding prefix operations on the front. Note that a *Schema_Ref* cannot be distinguished reliably in the lexer without more information, so PRECC's context dependent lexing is useful here.

```
@ Schema_Exp_U  = L_OPENBRACKET Schema_Text L_CLOSEBRACKET
@                                                 /* [ foo : fie | fee ]         */
@               | L_LNOT Schema_Exp_U             /* \lnot \lnot bar             */
@               | L_PRE  Schema_Exp_U             /* \pre \pre \pre bar          */
@               | L_OPENPAREN Schema_Exp L_CLOSEPAREN
@                                                 /* (((( bar ))))               */
@               | Schema_Ref                      /* foo [ fie ] [ fee / fum ]   */

@ Schema_Text   = Declaration [ L_VERT Predicate ]  /* [ foo:fie | fee ]         */

@ Schema_Ref    = Schema_Name Decoration [ Gen_Actuals ] [ Renaming ]
@                                                 /* foo [ fie ] [ fee / fum ]   */
```

A singleton *Renaming* has been reported to us as hard to distinguish from a simple division expression for a *Gen_Actuals* in *yacc* parsers. It might be a *Gen_Actuals* but for the presence of real *Gen_Actuals* in a *Schema_Ref*. When they are absent, the parse is ambiguous. PRECC has the same difficulty as *yacc* here because the concrete representations can be identical, not merely similar. Only hard semantic information could resolve what "`foo[fie/fum]`" means. If "`fie`" is a previously seen *Decl_Name* and it was first seen in schema "`foo`", then the interpretation may be resolved in favour of a *Renaming* rather than a *Gen_Actuals*.

```
@ Renaming      = L_OPENBRACKET                   /* [                           */
@                     Decl_Name L_SLASH Decl_Name /*          fee / fum          */
@                     { L_COMMA Decl_Name L_SLASH Decl_Name }*
@                 L_CLOSEBRACKET                  /* ]                           */

@ Declaration   = Basic_Decl { L_SEMICOLON Basic_Decl }*

@ Basic_Decl    = Decl_Name { L_COMMA Decl_Name }* L_COLON Expression
@                                                 /* foo , fie : fee             */
@               | Schema_Ref

@ Predicate     = L_FORALL Schema_Text L_AT Predicate
@                                                 /* \forall foo @ bar           */
@               | L_EXISTS Schema_Text L_AT Predicate
@                                                 /* \exists foo @ bar           */
@               | L_EXISTS_1 Schema_Text L_AT Predicate
@                                                 /* \exists_1 foo @ bar         */
@               | L_LET Let_Def { L_SEMICOLON Let_Def }* L_AT Predicate
@                                                 /* \let foo == fie @ bar       */
@               | Predicate_1
```

Note that *L_TRUE* and *L_FALSE* have to be distinguished from identifiers by the lexer. An identifier is nominally a sequence of alphanumeric characters, but it should not take the form `true` or `false`. Since PRECC would normally only use context to disambiguate the interpretation, there is a priori a danger of seeing `true` as an identifier in a context when an identifier is expected.

Care must therefore be taken to ensure that under no circumstances will `true` or `false` match on an identifier. The lexer will therefore have to specify an identifier more strictly than simply an alpha-numeric sequence. In the productions below, *L_TRUE* and *L_FALSE* are tested before *Schema_Ref*s, which avoids one ambiguity, but other opportunities for confusion exist and the lexical distinction is necessary.

```
@ Predicate_1   = Predicate_2 [ L_IMPLIES Predicate_1 ]
@                                             /* foo \implies   bar     */

@ Predicate_2   = Predicate_U                 /* foo                    */
@                 { { L_LAND | L_LOR | L_IFF }  /* \land                */
@                   Predicate_U               /* bar                    */
@                 }*                          /* \lor more              */

@ Predicate_U   = Expression { Rel Expression }*     /* foo = fie \in fee    */
@               | Pre_Rel_Decor Expression    /* foo' bar               */
@               | L_PRE Schema_Ref            /* \pre foo [ fie ]       */
@               | L_TRUE                      /* true                   */
@               | L_FALSE                     /* false                  */
@               | L_LNOT Predicate_1          /* \lnot foo              */
@               | L_OPENPAREN Predicate L_CLOSEPAREN /* ( foo )         */
@               | Schema_Ref                  /* foo [ fie ]            */

@ Rel           = L_EQUALS | L_IN | In_Rel_Decor
@               | L_INREL L_OPENBRACE Ident L_CLOSEBRACE

@ Let_Def       = Var_Name L_EQUALS_EQUALS Expression
```

At this point, the script becomes a little more interesting. The ZRM grammar defines the precedences of operators separately, but that shortcut does not exist for PRECC. Instead, we have to write out the grammatical constructions in an explicitly layered fashion. Our script here adds an extra production, *Expression_0* to capture the most weakly binding constructs, and then descends into the *Expression* production implicitly defined in the ZRM (this kind of restructuring turned out not to be necessary for schema expressions and predicates because the presentation in the ZRM is already sufficiently structured for our purposes).

```
@ Expression_0  = L_LAMBDA Schema_Text L_AT Expression
@               | L_MU Schema_Text [ L_AT Expression ]
@               | L_LET Let_Def { L_SEMICOLON Let_Def }* L_AT Expression
@               | Expression

@ Expression    = L_IF Predicate L_THEN Expression L_ELSE Expression
@               | Expression_1
```

To make the script a little neater, the layers are generally each split into two parts here, a front and a back. For example, an *Expression_1* consists of a front part – an *Expression_1A* – optionally followed by a back part – an infix operator preceding more level-1 expression syntax.

An *In_Gen* binds most weakly. The following is the recursive form for a sequence of sub-expressions (of type 1A) separated by *In_Gen* symbols. The latter are usually *L_INFIX* lexemes but may be any *L_WORD* at all that has been registered by a "`%%ingen`" directive. Their recognition will be discussed later in this section.

```
@ Expression_1  = Expression_1A [ In_Gen_Decor Expression_1 ]
```

An *L_CROSS* binds next most weakly.

```
@ Expression_1A = Expression_2 { L_CROSS Expression_2 }*
```

The generic class *In_Fun* binds next most weakly. These are generally *L_INFIX* tokens but it also is not a fixed class. Any *L_WORD* that has been registered by a "`%%inop`" directive will be recognized here. Moreover, there are distinct binding powers (from one, most binding, to six, least binding). To handle the bindings, we invoke a generic construct, *binex(l,m,e,s)*, which matches sequences of expressions *e* separated by binary operators *s*. The binding powers of the operators can vary between large *l* (in this case, one) and mild *m* (in this case, six).

```
@ Expression_2  = binex(1,6,Expression_2A,In_Fun_Decor)
```

The *binex* construction is defined as follows. The "`binpow($x)`" function looks up the key "`$x`" and returns the binding power of the operator it represents. That is recorded in the global table that is incremented by each "`%%inop`" directive. The *binex* first looks to the mildest level of binding, *m*, and tries to match against a sequence at that level. Then it descends to tighter levels of binding.

```
@ binex(l,m,e,s)= )l> m( e                 /* atomic expression  */
@               | )l<=m( binex(l,m-1,e,s)  /* expression level m */
@                        {s\x )binpow($x)==l( binex(l,m-1,e,s)}*
```

We have arrived at tightly binding prefix operators such as *L_POWER*. The generic class *Pre_Gen* binds as tightly, but is defined dynamically. It consists of *L_WORD* tokens that have been marked via an earlier "`%%pregen`" directive.

```
@ Expression_2A = L_POWER Expression_4
@               | Pre_Gen_Decor Expression_4
@               | L_HYPHEN Decoration Expression_4
@               | Expression_4 L_LIMG Expression_0 L_RIMG Decoration
@               | Expression_3

@ Expression_3  = Expression_4+
```

Note that the *Expression_3* clause really has to appear after the other clause starting with *Expression_4*. A single *Expression_4* may either be taken as a valid *Expression_3* or as an instance of the other clause in which the later part is missing. There is no problem if the whole parse is successful, but in case of error confusion will result as to its location. It is safer to put the clause which has a distinguishing token – the *L_LING* token – first.

There is also some inefficiency introduced here, in the form of an occasional double parse of the leading *Expression_4*. But the inefficiency is probably justified in order to be able to continue to follow the *f*UZZ grammar layout closely.

The treatment of function applications may not be immediately obvious from this syntax description. They appear in the above as an arbitrary non-zero sequence of expressions in *Expression_3*, with no explicit separators between them.

Expression_4 is being split into a two-part description here purely because of the convenience of isolating the possible postfix operator constructs in a single production.

```
@ Expression_4  = Expression_4A [ L_POINT Var_Name
@                               | Post_Fun_Decor
@                               | L_BSUP Expression L_ESUP
@                               ]
```

For brevity, a single name for the 'list of expressions' construct is introduced here:

```
@ Expressions   = Expression { L_COMMA Expression }*
```

There is no real reason to split up the level-4 expressions into an *_A* and a *_B* part, apart from the aesthetic preference for shorter lists of alternates in productions.

```
@ Expression_4A = Var_Name [ Gen_Actuals ]
@               | Number
@               | Set_Exp
@               | L_LANGLE [ Expressions ] L_RANGLE
@               | Expression_4B

@ Expression_4B = L_OPENPAREN Expressions L_CLOSEPAREN
@               | L_LBAG   [ Expressions ] L_RBAG
@               | L_THETA Schema_Name Decoration [ Renaming ]
@               | L_OPENPAREN Expression_0 L_CLOSEPAREN
@               | Schema_Ref
```

Set expressions are reputedly very difficult to handle in *yacc*-based parsers because a list of set elements may be confused with a list of of variables of the same type – both are separated by commas – until a trailing *L_COLON* or *L_CLOSESET* is discovered. With an infinite lookahead parser like PRECC this is not a problem.

There is a note in the ZRM that points out that the case of a singleton schema name is also ambiguous: i.e., is "`{S}`" to be taken as a set display, or should it be interpreted as a set comprehension, viz. "`{S | \Theta S}`"?

```
@ Set_Exp       = L_OPENSET [ Expressions ] L_CLOSESET
@               | L_OPENSET Schema_Text [ L_AT Expression ] L_CLOSESET

@ Ident         = L_WORD Decoration

@ Decl_Name     = Op_Name | Ident

@ Var_Name      = L_OPENPAREN Op_Name L_CLOSEPAREN | Ident

@ Op_Name       = L_UNDERSCORE In_Sym_Decor L_UNDERSCORE
@               | Pre_Sym_Decor L_UNDERSCORE
@               | L_UNDERSCORE Post_Sym_Decor
@               | L_UNDERSCORE L_LIMG L_UNDERSCORE L_RIMG Decoration
@               | L_HYPHEN Decoration

@ In_Sym        = In_Fun | In_Gen | In_Rel

@ Pre_Sym       = Pre_Gen | Pre_Rel

@ Post_Sym      = Post_Fun

@ Decoration    = Stroke*
```

We have added the following definitions. The annotations pass on the attribute attached to the root part of these terms as the attribute associated with the whole compound. It may be examined higher up in the parse.

```
@ In_Sym_Decor  = In_Sym\x   Decoration  {@ $x @}

@ Pre_Sym_Decor = Pre_Sym\x  Decoration  {@ $x @}

@ Post_Sym_Decor= Post_Sym\x Decoration  {@ $x @}
```

Here is one point at which a *yacc* parser may fail (depending on how the context is introduced). There is a potential reduce/reduce clash over whether to jump into a *Gen_Formals* or into a *Gen_Actuals* at the closing bracket. A *Gen_Formals* can look just like a *Gen_Actuals*. We have one report of this problem from a correspondent working with *yacc*.

```
@ Gen_Formals   = L_OPENBRACKET Ident { L_COMMA Ident }* L_CLOSEBRACKET

@ Gen_Actuals   = L_OPENBRACKET Expression { L_COMMA Expression }* L_CLOSEBRACKET
```

We have now covered everything except the atomic *lexemes* of the parse tree. All that remains is to pick up these lexemes from the lexer. Note the inherent ambiguity of the scheme given in the ZRM. Context alone dictates the interpretation of the lexemes at the parser base-level and we have had to add semantic checks here in order to test whether or not the token has been added to the appropriate table (by a "%%" directive).

```
@ In_Fun        = L_WORD\x  )is_inop($x)(    {@ $x @}

@ In_Gen        = L_WORD\x  )is_ingen($x)(   {@ $x @}

@ In_Rel        = L_WORD\x  )is_inrel($x)(   {@ $x @}

@ Pre_Gen       = L_WORD\x  )is_pregen($x)(  {@ $x @}

@ Pre_Rel       = L_WORD\x  )is_prerel($x)(  {@ $x @}

@ Post_Fun      = L_WORD\x  )is_postop($x)(  {@ $x @}

@ Stroke        = L_STROKE

@ Schema_Name   = L_WORD

@ Number        = L_NUMBER
```

We have also added the following definitions. Again, the annotations are in order to preserve the attribute attached to the root part of these terms as the attribute associated with the whole compound. It may be needed at a higher level.

```
@ In_Fun_Decor  = In_Fun\x   Decoration   {@ x @}

@ In_Gen_Decor  = In_Gen\x   Decoration   {@ x @}

@ In_Rel_Decor  = In_Rel\x   Decoration   {@ x @}

@ Pre_Gen_Decor = Pre_Gen\x  Decoration   {@ x @}

@ Pre_Rel_Decor = Pre_Rel\x  Decoration   {@ x @}
```

```
@ Post_Fun_Decor= Post_Fun\x Decoration   {@ x @}
```

That is the end of the parser description. A slightly abbreviated lexer description follows.

4 The Lexer Script

As discussed in Section 1, PRECC can handle two-level grammars, and there is nothing very unusual in using the same utility to handle both lexical and parsing phases. PCCTS [11] can do this too. The convenience is also an efficiency: automaton based lexers such as *lex* [9] and the newer *flex* are slow, usually taking up the majority of the parse time. Extending a relatively efficient parsing mechanism down to the character level can speed up the parsing process.

This lexer definition is independent of the parser definition (and vice-versa), except in that the parser expects distinguishing concrete information to be passed up as an attribute along with *L_WORD*. The lexer is not dependent on the parser. The tables that the parser constructs when it sees "`%%`" directives in the text are of relevance to the parser alone. The lexer could be rewritten to accept plain ASCII rather than LaTeX without affecting the parser.

We are following the definitions in the ZRM closely, but a few extra definitions help. The ZRM is not explicit on the following points: white space, parser directives, and LaTeX free forms that may be used as identifiers. We define *ws* ('white space') to consist of spaces, tabs, newlines, and also a LaTeX comment – a percent sign followed by arbitrary characters up to and including the newline.

To accommodate the use of "`%% `" at the beginning of a line that should be scanned by the parser but not be seen by LaTeX, that is treated as white space too. It would be easier to filter it out at a lower level, however. There are complications caused by the present approach when it comes to recognizing where ordinary text (with embedded LaTeX comments) ends and where a "`%%`" directive begins. A three-level grammar would provide a more efficient solution.

The space and tab characters are defined separately as *ws1* below. This is so that they can be picked out as the separators for "`%%`" directives, which cannot contain line breaks and for which other common LaTeX spaces may also be significant. The LaTeX tilde space and other standard LaTeX spaces fall in the (fuller) *ws* definition. More LaTeX symbols should be registered as white space through the use of `%%ignore` directives in the text, though that is a point of difficulty with our approach. Allowing analyzed LaTeX words as white spaces makes the definition recursive at the lowest level, incurring a performance penalty. We content ourselves with the standard spaces in the following fixed definition.

```
@ ws1   =         <' '> | <'\t'>

@ ws    =         ws1
@       |         nl    | <'%'> ?* nl
@       |         ^ <'%'> <'%'>  ws1
@       |         <'~'> | <'\\'> { <';'> | <','> | <'!'> }
```

To streamline the presentation here, we define a high level construct designed to match against a given string of characters. The *key0(*`"foo"`*)* construct, for example, will match the string of characters '`foo`', and saves us from writing out *¡'f'¿ ¡'o'¿ ¡'o'¿* in full in the grammar. Note that PRECC supports ANSI C and that the string (`w`) which appears in the construction (below) is a C string; that is, really the address of the first byte in the string. The C string physically occupies a contiguous sequence of bytes in memory terminated with a null byte. "`!*w`" is the C expression for 'string `w` is the empty string' – i.e., its opening byte is the null byte. This predicate is placed inside out-turned parentheses in the production below, which signify a guard condition in PRECC. "`*w`" is the C expression returning the opening byte in the string `w`, which may be non-null or null, so it stands for 'string `w` is not the empty string', and "`w+1`" is the C expression denoting the tail of the string `w`.

```
@ key0(w)=         )!*w( /* empty */
@        |         < *w> key0(w+1)
```

In detail, this construct accepts an empty input sequence when the parameter `w` is the empty string. If the parameter is a nonempty string, on the other hand, then it matches an incoming character equal to the opening byte of `w`, and then recursively matches the tail of `w`. In other words, the construct matches incoming character sequences against its parameter `w`.

We ignore trailing white space on a keyword match by using the construct *key(*`w`*)* instead of *key0(*`w`*)*. In order that the attribute attached be the string representing it, rather than (by default) derived from the white-space component, the definition contains the explicit attachment "`{@ w @}`". That makes "`w`" into the attribute attached to this construct.

```
@ key(w)=          key0(w) ws*  {@ w @}
```

A variant is *key1(*`w`*)*, which allows only trailing spaces and tabs.

```
@ key1(w)=         key0(w) ws1* {@ w @}
```

We follow the lexer macros given in the ZRM exactly here. The parenthesized names below denote calls to the standard C library functions "`isdigit`" and "`isalpha`" with the incoming character as their argument.

```
@ digit =          (isdigit)

@ letter=          (isalpha)

@ ident =          letter { letter | digit | <´\\´> <´_´> }*
```

Care has to be exercised since the ZRM lexical descriptions overlap. Here, for example, infixes are prevented from matching on an equals sign (or a double equals), which should be passed up to the parser as a *L_EQUALS* token, by the simple expedient of making an equals sign an optional first component of the lexeme. Now an equals sign on its own will not match against an infix because more input is expected after the initial optional equals sign. The *f*UZZ definition allows an equals sign to be passed as an *L_INFIX* instead.

```
@ infix =          [ <´=´> [ <´=´> ] | <´,´> ]
@                  { <´+´> | <´-´> | <´*´> | <´.´> | <´=´> |
@                    <´<´> | <´>´> | <´,´>
@                  }+  /* not ´=´ or ´,´ or ´==´ alone */
```

(Note, however, that a treble equals sign, for example, is a perfectly good match).

The ZRM does not tell us how to capture LaTeX constructs and in fact it is impossible; LaTeX is a mutable language. Here we make an attempt at recognizing as much as may be practical for the contexts in which we expect to encounter LaTeX, in schema headers, and so on. A LaTeX construct usually consists of a backslashed sequence of letters optionally followed by more of the same inside curly brackets (this definition excludes combinations like "`\!`", which will be seen as white space). To make sure that the initial backslashed sequence is not one of the reserved keywords such as "`\Delta`", however, an optional match against a list of (non-backslashed) *keywords* is forced after the backslash. Something like "`\Delta`" will then be rejected because the option matches, but there is no succeeding nonempty sequence of letters. On the other hand, something like "`\Deltafoo`" will match here.

```
@ latex =          <´\\´> [ keyword ] letter+   /* but not a keyword! */
@                  ws*
@                  [ <´{´> ws* { word ws* }* <´}´> ]
```

The LaTeX description above can obviously be improved. The authors have chosen to leave the final solution for later and safer hands.

The following are definitions for generic patterns used in the 'terminal symbols' given in the ZRM grammar. A priority should be restricted to the range 1–6 here, but, in the interests of clarity, we will allow any single digit character. The numerical value of the digit becomes its attached attribute.

```
@ number=          digit+

@ stroke=          <´?´> | <´!´> | <´_´> digit

@ priority=        digit\x  {@ $x - ´0´ @}     /* integer attribute */
```

The ZRM specifies that an infix really should be an alternate in *word*, but there are difficulties. It seems that this would make the parser see infixes and identifiers as interchangeable. The difficulty is resolved at the parser level by accepting only appropriately registered words as infixes. In practice, the registration occurs via "`%%`" directives in the parsed text. Without being registered, words will not be recognized as infixes at the parser level and it is safe to allow infixes as an alternative in words here. The (ZRM) design just leads to more parser level testing. The parser effectively is given a monolithic token class *L_WORD* which it then breaks down into subclasses again, using tabulated information.

```
@ word  =          ident | latex | infix
```

The *nl* macro captures a literal end-of-line condition so that an action may be attached (such as incrementing a line count for display on error). All other productions refer to *nl* and not the PRECC ground construct '*$*'.

```
@ nl     =         $
```

Now for the list of keywords (without prefixed backslashes) which have to be distinguished from LaTeX constructs. The real version of this script achieves this using a binary search down a tree, but, for brevity, that approach will not be followed here. The *keyword* production can be a plain list of alternates. The full list is rather long (thirty nine entries) so the contents will be indicated by an ellipsis below:

```
@ keyword =  key0("Delta") | key0("Xi") | ... | key0("where")
```

As mentioned earlier, *L_COMMENTCHAR*s can be essentially anything. By the time the parser comes to call for one, it has exhausted all other possibilities. But how can the parser know when a sequence of *L_COMMENTCHAR*s has come to a stop? Only if one of the valid openings for another top-level *Paragraph* is seen next. So we make sure that *L_COMMENTCHAR* cannot match the first character of another valid Z paragraph.

There are two sorts of other paragraph. Those that begin with "`\begin{`" and contain a Z specification, and those that begin with "`%%`" and contain a directive. We make sure not to match these by (optionally) scanning over their beginning sequence and then demanding a nonstandard ending. The backslash in "`\begin{zed}`", for example, will be rejected as a *L_COMMENTCHAR* because a match on "`\begin{zed`" (no closing brace) will happen first, but then something which is *not* a close brace is required.

Moreover, the white-space that is allowed to follow a keyword will rightly gobble any following LaTeX comment (beginning with a percent sign), but an opening "`%%`" directive on the next line could be matched too. If a percent were matchable as part of a *L_COMMENTCHAR*, then such a directive would be partially eaten and what remained would look like a LaTeX comment and be matched as white space. So we forbid a final percent as well as a close brace (after a keyword). Real LaTeX comments are matched as explicit white-space, and a single close brace on its own is matched explicitly.

```
@ L_COMMENTCHAR  = { [ key("\\begin") L_OPENBRACE
@                      { key("zed")
@                      | key("axdef")
@                      | key("schema")
@                      | key("gendef")
@                      }
@                    ] (not_percent_nor_closebrace)
@                  |   <´}´> | ws
@                  } ws*
```

We have handled nearly everything now. The ZRM is being followed almost exactly again. *L_WORD*s may optionally contain an opening "`\Delta`" or "`\Xi`", separated by some white space from the following word. We always return the attribute of the latter, using the *cache* construct explained in the following paragraph to establish some value (here, a unique integer key) that will identify the token uniquely now and when it is seen again.

```
@ L_WORD =         [ { key0("\\Delta") | key0("\\Xi") } ws+ ]
@                  cache(word)\x ws* {@ $x @}
```

As remarked, we have to compute and attach an identifying integer key for infixes, idents and so on. To do that, we need to buffer incoming characters and then compute a unique value from them. There are good and bad ways of doing this, and to save space here, we use a bad one.

The proper thing to do is to pass a local buffer into the pattern matches as an inherited attribute, explicitly make each incoming character write into the buffer, then pass on the latter part of the buffer to a successor. The ("improper") trick used here is to steal the lexers built-in buffer and take advantage of the fact that it gets written to automatically via implicit side-effects. That allows us to avoid cluttering up the script with annotations. We use a *dummy* definition whose sole purpose is to return the current buffer position "`pstr`" as an attribute. When we reach the end of a sequence of characters that comprise an interesting lexeme, we use another *dummy* entry to return the then buffer position. The difference between the two positions is the length of the string comprising the lexeme and we send the first buffer position and the length to a function “ukey” which looks up or computes a new (unique) integer key for the lexeme. The generic construction is called *cache*.

```
@ dummy    =      {@ pstr @}   /* current input buffer pointer */

@ cache(p) =  dummy\begin p dummy\end
@             {@ ukey($begin,$end-$begin) @}
```

The PRECC lexer input buffer is never emptied until an action attached to the parse is executed, so it is safe to refer to it here. The only actions specified for this grammar occur after a directive is encountered in the script. That means that the buffer will not be emptied during a scan of any of the patterns we are interested in.

```
@ L_STROKE =      stroke ws*

@ L_INFIX =       cache(infix)\x  ws*   {@ $x @}   /* key attribute */

@ L_NUMBER =      cache(number)\x ws*   {@ $x @}   /* key attribute */

@ L_PRIORITY =    priority\x      ws*   {@ $x @}   /* integer attr. */

@ L_SYMBOL =      cache(word)     ws1*  {@ $x @}   /* key attribute */
```

The remainder of the script just consists of explicit simple token matches;

```
@ L_ELSE =        key("\\ELSE")

...

@ L_EQUALS =      key("=")

@ L_PERCENT_PERCENT_INOP =    ^ key1("%%inop")

...

@ L_PERCENT_PERCENT_TAME =    ^ key1("%%tame")

@ L_ENDLINE              =     nl ws*
```

Compound keywords such as “\begin{gendef}” have some slackness built into their pattern matches, allowing white space where appropriate.

That concludes the lexer specification. The lexer and parser specifications have to be run through the PRECC utility to generate ANSI C; then the C code is compiled to object code using an ANSI compiler such as GNU’s *gcc* and linked to a PRECC kernel library.

5 Observations

The executable we have appears to be successful, in that it detects errors in those Z scripts that have obvious errors in them and passes those that look correct, However, in the absence of a comprehensive test suite for the ZRM grammar, little more than that can be ascertained. It is to be expected that the current Z standardization effort [6] will generate at least a *yacc*-based ‘official’ standard in the long run, necessarily incorporating a set of tests.

We have somewhat informally tracked errors over the last several months via a log of correspondence. A priori, there are errors of the following kinds that we may have expected to see:

1. errors in the original BNF;
2. errors of understanding of the original BNF;
3. errors of transcription of the BNF to PRECC;
4. errors of (automatic) translation by PRECC to C;
5. errors of compilation of C.

We have found no errors of the first kind, although there are omissions. Generally, we have found the "90/10" rule to apply. Coding up the BNF for PRECC proceeded very quickly, for example, (less that 2 person-days), but adding on missing parts to reach the point where the parser could run took another ten person-days. In the end, covering 90% of the grammar indeed has taken only at most 10% of the time, and the remainder has taken up a disproportionate share. Where the grammar has been informally specified or unspecified to begin with has proved to demand by far the hardest work. Or, to put it another way, the work embedded in the specification that we started from saved us much time.

The hardest remaining questions to settle were the forms of LaTeX to be recognized by the lexer, and what exactly constitutes white space. These apparently trivial questions are still not completely answered after at least some six months of on/off collaborations and testing. The present pattern match for LaTeX is simplistic – anything beginning with a backslash possibly followed by any sequence of such things in braces. But it may not be simple enough. Perhaps brace nesting is all that really needs to be examined here. The problem with white space for Z is that it may be defined dynamically as the parser hits more "`%%ignore`" directives, and it is potentially very inefficient to treat this as anything other than a side-effect on a primary input filter. We have not yet found anything more acceptable.

There are also some disagreements on binding priorities (for example of *L_IFF* over *L_IMPLIES*) between different versions of the published grammars, and we have resolved these in favour of the copy we possess over the copy that correspondents are referring to.

The omission from the original BNF with the biggest impact is a description of how to clearly separate out infixes, identifiers and the like. The original document (as does this PRECC script, following it) specifies many terminals that the parser receives from the lexer as *L_WORD*s, without distinction. *L_WORD*s can match *infix* or *ident* or other kinds of specific pattern, but the parser asks for just an *L_WORD* when it is looking for both *Var_Name* and *In_Gen*, for example. Following this BNF originally led to what were plainly identifiers being seen as infixes, and vice-versa. It took some weeks to recognize the nature of the problem and then a matter of a person-week to fix. This required adding in the treatment of the "`%%`" directives detailed here. These contribute to a parser-level reclassification of *L_WORD*s back into subclasses again.

The fact that Z scripts consist chiefly of non-Z parts was also not explicit in the original BNF. But this omission was recognized early, and the treatment of textual comments as an extra kind of top-level *Paragraph* was the response.

Errors in understanding of the original BNF are certainly possible, but we have not had to understand too much because the modifications for a PRECC script are relatively minor and can be carried out without more than local understanding. PRECC has compositional semantics. There have been difficulties in determining whether or not one or two problematic test scripts that pass the PRECC parser really should be accepted, even though they contain unusual constructions, because then we have not been able to rely on the common pool of knowledge on acceptable Z constructions and have had to think hard about the original BNF. The difficulty lies in tracing through the alternatives in the BNF to see if a given interpretation of a concrete sequence of symbols is valid. This is very hard for a human to do unaided, but it has been noticed that some people have considerably greater skill than others in this regard! Thanks to that sort of input, we have not noted any residual errors of comprehension since testing began in earnest.

There were two errors of transcription of the BNF for PRECC. One was an omission of one '|' symbol in the definition of *Stroke*. The other was the initial coding of most parser symbols as *L_WORD*s, as described above, and a removal of the *infix* alternative from the definition of *L_WORD* in compensation. These errors were eventually corrected, as reported above.

There were also several incorrect codings in the original PRECC script. "Bugs", in other words! It was not initially realized that the lexer would have to detect the start of valid Z *Paragraph*s in order to avoid running through them as though they continued a prior text commentary paragraph. The definition of a *L_COMMENTCHAR* was initially incorrect and the mistake was not detected because the test scripts did not contain much text. Four or five more such bugs were uncovered over several months. The errors were corrected as collaborators reported them, but the coding time involved is minor – of the order of minutes.

As regards the fourth category of error, bugs in PRECC itself are rarely uncovered nowadays. The rate is about one every six months, and they are minor, meaning that they affect rarely exercised functionalities. And there is not much of that because PRECC is based on a simple design compounded from only two or three basic components, and all of these tend to be exercised equally. The only significant bug corrected in this period involved a buffer overflow in the PRECC parser generator, which did not affect PRECC client parsers.

Bugs in the C compiler (which is a much more complex application) are discovered more frequently. Indeed, the development of PRECC over the years has exposed several bugs in widely available proprietary compilers. The HP proprietary C compiler for the HP9000 series used to compile incorrectly functions with a variable number of parameters of size less than the size of an integer. The Borland Turbo C 3.0 compiler silently compiled a switch statement followed by a register variable access incorrectly. The IBM AIX C compiler would not accept variables which were declared using an alias (`typedef`) for a function type. And so on. For the most part we have relied on GNU's *gcc* 2.5.8 and 2.7.0 for i386 architectures, and 2.3.1 and 2.5.8 for suns. No differences in behaviour are detected between all these.

Testing the grammar description here has proved a difficult undertaking. One might like to build the grammar part by part and test each part against what it ought to do, then put the whole together. The compositional semantics of PRECC would guarantee that all the parts compose correctly. But there is no point in testing small parts individually because the grammars they express are easily comprehended in terms of the terminal units and it is known that PRECC correctly puts together grammars from parts. A small portion of the grammar will behave exactly as we expect it to. The difficulty lies with the integration of parts, despite the compositionality. The complexity of the grammar description as a whole can defeat the capacity of the human mind.

For example, a Z script has a well-defined structure consisting of a series of *Paragraph*s. One kind of paragraph matches textual commentary and other kinds of paragraph capture the different kinds of Z "boxes". Another kind of paragraph matches a "`%%`" directive. White space may separate paragraphs and may include LaTeX comments, which start with a percent. Two percent symbols followed by a space at the beginning of a line are also white space. White space may also appear within paragraphs. Text paragraphs cannot overrun the beginning of Z boxes or of "%%" directives. Can anyone predict, from that description, whether a text paragraph may overrun a "%% " immediately following it? One may think not, but the detailed specifications of each kind of paragraph would have to be examined in order to settle the point, and correlating what three different complex specifications say simultaneously is difficult. In classic style, we can be certain that the parser implements what we have specified, but not intrinsically certain that we have specified what we meant. The difficulty is compounded for parts of the grammar that originate with other authors. PRECC does have a well defined axiomatic semantics [3] that permits such questions to be settled absolutely, but the difficulty of doing the reasoning remains. Making the part of the grammar that originates with us as clear as possible is the only defense, but, as indicated, even residual pockets of complexity can spread uncertainty. We are helped by the nature of the PRECC scripting language, which is both clear and expressive. We have been able to make use of meta-grammatical constructs to structure difficult portions of the grammar, such as the part dealing with expressions containing separators of different binding powers, and would have used more meta-constructs if we had not been attempting to follow the form and layout of the original BNF so closely.

How does the development path here compare with a *yacc*-based one? A *yacc* parser requires much more detailed knowledge of the parser semantics, because the one-token lookahead restriction is quite severe. It is doubtful that a stateless *yacc* grammar for Z can be built. We would expect to encounter many, many more errors of transcription of the original BNF for *yacc* than for PRECC. The task of rewriting the grammar is very substantial.

In a sense, there would also be errors of translation to C (code for a finite state automaton) by *yacc*. A grammar script often gives rise to subtle violations of the em yacc one-token lookahead restriction that are reported as shift-reduce or reduce-reduce conflicts. It is very rare to find a *yacc* script with no such conflicts in it. Shift-reduce conflicts are resolved in favour of "shifts" by default, which means that the parser will be more forgiving than the grammar specifies. Reduce-reduce conflicts may result in very unexpected behaviours. They occur when the one advance token that the parser has seen cannot determine which alternative to match in a grammar clause. It is possible to rewrite the grammar to produce the equivalent of a two- or three-token lookahead, but this entails much duplication. The resulting automaton might be very large, and there are several points in the Z grammar where long and sometimes unbounded lookaheads are required. For example, within an unboxed Z paragraph, an item may appear that starts "`foo[fie,fee,`" and continues until a closing "`]`" arbitrarily far ahead. If a "`==`" follows then "`foo`" should have been a *Var_Name*, but if a "`\defs`" follows, it should have been a *Schema_Name* instead. The information comes too late for a *yacc* parser.

6 Summary

This document has set out a concrete Z grammar for the publicly available compiler-compiler PRECC. The design follows the published ZRM grammar very closely. It is a two-level top-down description. Communication between the top- (parser) and bottom- (lexer) levels is minimal, except for the passing of context information from top- to bottom-level, which happens automatically, and one mode change that the top-level can cause in the bottom-level. This allows the parser to notify the lexer of new white space. Other mode changes, such as the registration of new infix symbols, are entirely confined to the parser level. The lexer only has to pass upwards a unique key for each new *L_WORD* token it encounters (additionally, but trivially, information on the number represented by an integer token has at one point to be passed up). This means that the parser level is decoupled from the concrete representation of tokens. It would be easy to reconfigure this description for an ASCII rather than LaTeX-based description, for example, because only the lexer would need to be changed.

The script shown here differs from the complete implementation script only in missing out some repetitious elements from the lexer description (noted in the text and indicated by an ellipsis in the specification), omitting the standard C language "`#include`" directives for the appropriate library header files ("`ctype.h`", "`stdlib.h`", etc.), and omitting the C function definitions for certain attached actions.

The full script is available on the World-Wide Web under the following URL (uniform resource locator):

`http://www.comlab.ox.ac.uk/archive/redo/precc/zgram.tar.gz`

The version 2.42 PRECC utility for UNIX and DOS is available by anonymous FTP from `ftp.comlab.ox.ac.uk`, in the `pub/Programs` directory. The file names all begin with the sequence '`precc`'. Version 2.42 for DOS is available worldwide from many archive sites, including `garbo.uwasa.fi` and mirrors. An *archie* search for `precc232.zip or precc242.zip` should find the nearest copy for your location. General information on PRECC, with pointers to the latest on-line versions and papers, is available on the World Wide Web under the following URL:

`http://www.comlab.ox.ac.uk/archive/redo/precc.html`

Acknowledgements

We are grateful in particular to Doug Konkin (Department of Computer Science, University of Saskatchewan) for information on the behaviour of a *yacc* grammar derived from this specification, and for numerous valuable comments and discussions. Wolfgang Grieskamp (Department of Computer Science, Technical University of Berlin) has also contributed reports.

Jonathan Bowen was funded by the UK Engineering and Physical Sciences Research Council (EPSRC) on grant no. GR/J15186.

References

[1] J. P. Bowen and M. J. C. Gordon. A shallow embedding of Z in HOL. *Information and Software Technology*, 37(5-6):269–276, May–June 1995.

[2] P. T. Breuer and J. P. Bowen. A PREttier Compiler-Compiler: Generating higher order parsers in C. Technical Report PRG-TR-20-92, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK, November 1992.

[3] P. T. Breuer and J. P. Bowen. The PRECC compiler-compiler. In E. Davies and A. Findlay, editors, *Proc. UKUUG/SUKUG Joint New Year 1993 Conference*, pages 167–182, St. Cross Centre, Oxford University, UK, 6–8 January 1993. UK Unix system User Group / Sun UK Users Group, Owles Hall, Buntingford, Herts SG9 9PL, UK.

[4] P. T. Breuer and J. P. Bowen. A PREttier Compiler-Compiler: Generating higher order parsers in C. *Software—Practice and Experience*, 25(11), November 1995. To appear. Previous version available as [2].

[5] S. M. Brien. The development of Z. In D. J. Andrews, J. F. Groote, and C. A. Middelburg, editors, *Semantics of Specification Languages (SoSL)*, Workshops in Computing, pages 1–14. Springer-Verlag, 1994.

[6] S. M. Brien and J. E. Nicholls. Z base standard. Technical Monograph PRG-107, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK, November 1992. Accepted for standardization under ISO/IEC JTC1/SC22.

[7] P. Deransart and J. Małuszyński. *A Grammatical View of Logic Programming*, chapter 4, pages 141–202. The MIT Press, Cambridge, Massachusetts, 1993.

[8] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.

[9] S. C. Johnson and M. E. Lesk. Language development tools. *The Bell System Technical Journal*, 57(6, part 2):2155–2175, July/August 1978.

[10] L. Lamport. LaTeX *User's Guide & Reference Manual*. Addison-Wesley Publishing Company, Reading, Massachusetts, USA, 1986.

[11] T. J. Parr, H. G. Dietz, and W. E. Cohen. *PCCTS Reference Manual*. School of Electrical Engineering, Purdue University, West Lafayette, IN 47907, USA, August 1991. Version 1.00.

[12] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, January 1988.

[13] J. M. Spivey. *The* fUZZ *Manual*. Computing Science Consultancy, 34 Westlands Grove, Stockton Lane, York YO3 0EF, UK, 2nd edition, July 1992.

[14] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[15] Xiaoping Jia. *ZTC: A Type Checker for Z – User's Guide*. Institute for Software Engineering, Department of Computer Science and Information Systems, DePaul University, Chicago, IL 60604, USA, 1994.