PAPERS OF WARSAW UNIVERSITY/BIAŁYSTOK BRANCH
No 31 Humanities Vol. V
L – LOGIC

ZESZYTY NAUKOWE FILII UW W BIAŁYMSTOKU
Zeszyt 31 Humanistyka Tom V
DZIAŁ L – PRACE Z LOGIKI

STUDIES IN LOGIC, GRAMMAR, AND RHETORIC
I

EDITED BY
Witold Marciszewski, Jerzy Kopania

BIAŁYSTOK – 1980

CONTENTS

STUDIES IN LOGIC, GRAMMAR AND RHETORIC I
Warsaw University, Białystok Branch, 1980

ANDRZEJ TRYBULEC
Warsaw University, Białystok Branch

THE MIZAR LOGIC INFORMATION LANGUAGE

A logic information language /L.I.L., in short/ is a language of recording of mathematical texts, including the logical relationships between elements in it such as inferences between theorems, and so on. Our aim was the elaboration of a language, subsequently called MIZAR, the use of which would not depart too radically from accepted practices in mathematics. Such a practical L.I.L. has to be on the one hand sufficiently simple for automatic processing and, on the other, sufficiently rich for human use.

A L.I.L. of the type represented by MIZAR can be used for the following purposes:

a. for checking proofs submitted by students in mathematics teaching ;

b. in computer-aided proof-checking, particularly checking proofs of program properties ;

c. as an input language to a mathematics fact retrieval system ;

d. as a source language of semi-automatic machine translation, i.e. a translation from the L.I.L into a natural language

e. as an intermediate language in a machine translation system.

Much work in the field of L.I.L.'s has been devoted to the last topic /for instance, E.V.Padučeva/.

It must be noted that the analyser of a L.I.L. represents only a part of the system for most of the above-mentioned applications. For teaching and for computer-aided proof checking a supplementary module, known as proof checker, must be incorporated into the system. For teaching purposes it does not need extensive automatic proof checking facilities, but it must be "transparent" and must spend not much run time for inference checking. For the purposes of computer-aided proof checking, on the other hand, a "rich" system is preferable and run time limitations are not so essential, particularly when the system is being used interactively and the user can always break in, halt the checker and omit the justification of a statement. Into a retrieval system at least data bank access procedures must be incorporated; they would also be very useful in computer-aided proof checking. The proof checker can be discarded altogether in the case of machine translation.

Work on languages of this type has been underway for about ten years: one thinks for instance of Kaluźnin's paper [3] and of projects of de Bruijn [1] and Gluškov [3] . Recently, the projects of Postma [4] and Marinov have been described.

The MIZAR project is a continuation of work conducted for Płock Scientific Society in 1975 /implementation on an ODRA 1204 computer/. Later, the MIZAR project was implemented on the Cyber 72 computer using the PASCAL 6000 programming language. At first, a subset called MIZAR/QC has been implemented /see [6]/.

The syntax of MIZAR :

We use the BNF notation with slight modifications :

i. An interrogation point behind a conctruct denotes that it may be omitted.

ii. LIST (XXX, SEP) denotes (XXX SEP)* XXX and LIST

(XXX, SEP) denotes LIST (XXX, SEP) SEP XXX .

iii. An asterisk /resp. a plus sign/ used as superscipt denotes that the construct may be repeated an arbitraty number of times /resp. at least one time/ .

1. Expressions

```
expression: = packed-expression|
              packed-expression binary-operation packed-expre-
              sion
              unary-operation packed-expression
              associative-expression
packed-expression := " ( " expression " ) "|
              functional-expression|
              simple-expression
 functional-expression :=
              function-identifier:ao " (" expression-list " ) "
simple-expression := variable:ao | integer-constant
unary-operation := operation
binary-operation := operation₁
operation := operation₁ | "+" | "* "
expression-list := LIST  expression, ","
associative-expression :=
              plus-expression | asterisk-expression
plus-expression := LIST⁺  (packed-expression, "+")
asterisk-expression := LIST⁺  (packed-expression, " * ")

type-expression := type-identifier:ao ("of" expression-list )?
attribute-expression := attribute-identifier:ao
                                          ("of" expression-list ) ?
```

2. Sentences

```
sentence := packed-sentence |
          packed-sentence ("implies" | "iff") packed-sentence |
          conjunctive-sentence |
          disjunctive-sentence |
          universal-sentence |
          existential-sentence
packed-sentence := "(" sentence ")" |
          atomic-sentence |
          "not" packed-sentence
atomic-sentence := expression "is" attribute-expression |
          predicate-identifier:ao ("[" expression-list "]") ? |
          expression relation expression
disjunctive-sentence := LIST+ (packed-sentence, "or")
conjunctive-sentence := LIST+ (packed-sentence, "&")
universal-sentence := universal-prefix
                     ("holds" sentence |
                      existential-sentence
existential-sentence := "ex" being-list "st" sentence
universal-prefix := "for" being-list ("st restriction ) ?
being-list := (variable-list "being" type-expression ",")
           variable-list ("being" type-expression ) ?
variable-list := LIST (variable:do, ",")
restriction := sentence

proposition := (label:do ":") ? sentence
conditions := that LIST (proposition, "and")
```

3. Statements

```
statement := simple-statement |
             compound-statement
simple-statement := "then" ? statement-body ("by" label-list)?|
             statement-body justification
compound-statement := (label:do ":") ? "now reasoning  "end"
statement-body := proposition|
                  choice |
                  pred-clause |
                  take-clause |
                  reconsider-clause
choice := "consider" choice-list-1 "such" conditions |
     universal-prefix "consider" choice-list-2 "such"conditions
choice-list-1 := being-list
choice-list-2 := LIST (function-list "being" type-expression,
                       ",")
pred-clause := universal-prefix? "pred"
                 (Predicate-identifier:do |
                 variable:ao relation variable:ao)
                            "denotes" sentence
take-clause := "take" LIST ( take-list-1, ",")|
     universal-prefix "take" LIST( take-list-2, ",")
take-list-1 := variable-list "=" expression
take-list-2 := LIST (( function-identifier:do
                       operation variable:ao
                       variable:ao operation variable:ao), ",")
                                "=" expression
function-list := LIST ( function-identifier:do, ",")
reconsider-clause := "reconsider" LIST  rc-list-1, ","
```

```
universal-prefix "reconsider" LIST (rc-list-2, ",")|
rc-list-1 := LIST  variable:do("=" " ( "  expression-list " ) ")?
          ",")
                                "as" type-expression
rc-list-2:= LIST  (function-identifier:do
        ("="  " ( " LIST  function-identifier:ao, ",") ")  ")?,
         ",")
                                "as" type-expression
```

4. Justification

```
label-list := LIST (label:ao, ",")
 justification := "proof reasoning "end"|
                  "ref" reference|
                  "from" scheme-expression
reference := LIST  ( block-identifier:ao, "/" ) "/"  label+ao
scheme-expression := scheme-identifier:ao ( " ( "  label-list
                     " )  ") ?
```

5. Reasonings

```
reasoning := LIST(( assumption|
                    let-clause|
                    conclusion|
                    statement), ";")  ";" conslusion
conclusion := ( "thus" | "hence") proposition("by" label-list) ?|
                  "thus" proposition justification
assumption := "assume"   proposition   conditions
      "given" being-list  "such" conditions
let-clause := "let" LIST (variable-list "be" type-expression,
              ",")
```

("<u>such</u>" conditions)?

6. Definitions

6.1. Type definition

```
type-definition :=
    "type" type-identifier:do ("of" being-list ) ?
        ( "denotes" type-expression |
          "includes" variable:do "being" type-expression |
          "constists" "of" component-list) ("such" conditions ) ?
component-list := being-list
```

6.2. Function and/or attribute definition

```
definition := "definition"
                  LIST(( let-clause |
                         prime-definition),  ";")
                         "end"
prime-definition := definition-body ("by" label-list)   ? |
          definition-body justification
definition-body :=
       (variable:ao "is" attribute-identifier:do
                 "of" LIST   variable:ao, ","   ?
               "iff" sentence
```

6.3. Scheme definition

```
scheme-definition := scheme-head scheme-body
scheme-head := "scheme" scheme-identifier:do ";"
scheme-body :=
       "type" formal-type-list ";"  ?
       "predicate" formal-predicate-list ";"  ?
```

```
        ("function" formal-function-list ";") ?
        ("constant" formal-constant-list ";") ?
              formal-conclusion
        ("since" formal-assumptions-sequence) ?
              "proof"  reasoning "end"
formal-type-list := LIST (type+identifier:do, ",")
formal-predicate-list := LIST (predicate-identifier:do, ",")
formal-function-list :=
        LIST (function-list "being" type-expression, ",")
formal-constant-list := being-list
formal-assumptions-sequence :=
        LIST ( formal-assumption, ",")
formal-assumption := proposition
 formal-conclusion := sentence
```

7. Blocks

```
block :=  (block-identifier:do ":") ?
       "begin" LIST ( blòck-element, ",") "end"
block-element := statement |
                 definition |
                 scheme-definition |
                 type-definition |
                 theorem |
                 predeclaration |
                 block
theorem := "theorem"
                 (proppsition ("by" label-list)  ? |
                  proposition justification)
```

predeclaration := "<u>let</u>" LIST (variable:po, ",")

"<u>denote</u>" type-expression

8. Identifiers and constants

variable := identifier

label := identifier

block-identifier := identifier

type-identifier := identifier

scheme-identifier := identifier

attribute-identifier := identifier

function-identifier := identifier

predicate-identifier := identifier

identifier := letter (letter | digit)*

integer-constant := digit+

letter := A | ... | Z | a | ... | z

digit := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

References

[1] N.G. de Bruijn: "The mathematical language AUTOMATH, its usage and some of its extensions", Lecture Notes in Mathematics, Vol. 125, Springer Verlag, 1970.

[2] V.M.Gluškov: K postroeniju prakticeskogo formalnogo jazyka dla zapisi matematiceskich teorij, "Kibernetika", Nr. 5, 1972.

[3] L.A.Kałużnin: O języku informacyjnym matematyki, "Wiadomości Matematyczne", VII.2, 1964.

[4] S.W. Postma: FEA - A Formal English Subset for Algebra/ Assertions, "ACM SIGPLAN Notices", vol. 13,7, 1978.

[5] A.Trybulec : Informations-ligische Sprache MIZAR, "Dokumentation/Information", Heft 33, Ilmenau 1977.

[6] A. Trybulec: The MIZAR/QC/6000 Logic Information Language", ALLC Bulletin", Vol.6, No. 2, 1978*.

Andrzej Trybulec

Allatum est die 16 Octobris 1979

* This report is a revised and extended version of the text [6].