§1 Mizar Parser INTRODUCTION 1

Chapter 0

Introduction

1. We are trying to understand Mizar. So I am transcribing the source code into a literate program, following the order of compilation. Perhaps this "goes against the spirit" of literate programming, but it makes the most sense to understand what is going on for programmers.

We will begin with the esmprocessor.pas file, and all the dependencies needed to compile it. For clarity (or at least ease of reference) each "section" appearing in the table of contents corresponds to a different file. This discusses Mizar's source code as of Git commit 9e814a9568cfb44253d677e5209c360390fe6437 (dated 2023 October 11).

2. Files are chapters. We will organize the text by compiler dependencies. It makes sense to treat each file as a separate "chapter". With the exception of this introductory chapter ("chapter 0"), all future chapters are called "File n".

Just as Knuth's T_EX : The Program (Addison-Wesley, 1986) was organized into modules which are presented "bottom-up", each module is discussed and programmed "top-down", we shall try to do likewise. File n+1 can only depend on code appearing in Files 1 through File n.

There are natural ways to "cluster" the discussion in each File, which motivates the "section" and "subsections". Each section (but not subsections!) starts on a new page, written in sans serif bold prefixed with explicit an "Section". Subsections are written in sans serif bold prefixed with an explicit "Subsection", with vertical whitespace separting it. This chapter has two sections (one discussing the flow of Mizar, and the other enumerating observations and "to do" items).

3. Each chapter is written using numbered paragraphs, since we are using Donald Knuth's WEB to write a literate program. References will be made to the paragraphs. Index entries give the paragraph numbers associated with each entry. And even though I just used the term "paragraph number", they really group several paragraphs into a unit of writing.

Numbers are paragraphed *independently* of chapter, section, subsection. This is a quirk of WEB. This was how Mathematicians wrote texts back in Euler's day. We will refer to a paragraph by writing $(\S n)$ to refer to paragraph n. Again, this was the conventions used by Euler.

Each paragraph consists of three parts: the "text part" (informal prose written in English), the "macros part" (which introduces macros written in the WEB language), and the "code part" (which contains a pretty-printed snippet of PASCAL code). A paragraph may omit any of these parts. Thus far, all our numbered paragraphs have consisted of "text parts" only. The "code part" can have a name in angled brackets. If the name is missing, then it continues the previous chunk of code from the previous numbered paragraph.

- 4. The Mizar program is released under the GNU license. So let us place this license in one place early on. $\langle \text{GNU License 4} \rangle \equiv$
 - { This file is part of the Mizar system. Copyright (c) Association of Mizar Users. License terms: GNU General Public License Version 3 or any later version. }

This code is used in sections 23, 54, 77, 83, 89, 102, 115, 128, 182, 462, 463, 485, 487, 514, 554, 588, 647, 681, 719, 735, 850, 1193, 1199, and 1492.

2 INTRODUCTION Mizar Parser §5

5. Aside: Typography of "Modern" Pascal. We will be following the typographical style as found in Niklaus Wirth's Algorithms + Data Structures = Programs (Prentice-Hall, 1975) and Donald Knuth's TeX: The Program (Addison-Wesley, 1986). But there are a few typographical situations which requires thinking hard about, since "classical" PASCAL does not have object or inheritence (or unit modules).

First, we need to know that "modern PASCAL" differs from the PASCAL Knuth worked with, in several ways. Mizar uses "units" which are modules. We will need to format them for WEAVE. Note that §42 of WEAVE introduces the "ilk" of various syntactic classes (array-like, case-like, for-like, etc.) and §64 of WEAVE explicitly initializes all the reserved words for PASCAL.

The reserved words which WEAVE highlights and/or pretty prints: and, array, begin, case, const, div, do, downto, else, end, file, for, function, goto, if, in, label, mod, nil, not, of, or, packed, procedure, program, record, repeat, set, then, to, type, until, var, while, with, xclause. These are then grouped into 20 "ilk" classes (see §42 of WEAVE).

Documentation and tutorials frequently compare unit to program, so we should probably typeset it as such. The big question is whether the interface, implementation, and uses keywords are var -like or const -like. I ultimately decided for the latter.

We will treat **implementation** typographically as if it were a **const** because the **end** will not be indented properly otherwise.

```
format unit \equiv program
format interface \equiv const
format implementation \equiv const
format uses \equiv const
```

6. Objects appear in Free PASCAL, and they behave like records. There are also **constructor** and **destructor** functions.

```
format object \equiv record

format constructor \equiv function

format destructor \equiv function
```

7. **Primitive functions.** We have several primitive functions which should be formatted especially. For example, **shr** is an infix operator like **mod** or **div**. It corresponds to bitwise shifting right.

```
format shr \equiv div
```

8. Cases. Following Knuth's "TeX: The Program" (§4), we will use **endcases** to pair with **case**. The "default case" will be **othercases** (because **else** gets too confusing).

```
define othercases \equiv others: { default for cases not listed explicitly } define endcases \equiv \mathbf{end} { follows the default case in an extended case statement } format othercases \equiv else format endcases \equiv end
```

9. Debugging. There are conditional compiler directives for debugging purposes. Importantly, these *must* be printed to the source code when we invoke TANGLE.

```
define mdebug \equiv @\{@\&\$IFDEFMDEBUG@\}
define end\_mdebug \equiv @\{@\&\$ENDIF@\}
format mdebug \equiv begin
format end\_mdebug \equiv end
```

§10 Mizar Parser INTRODUCTION

10. Actually, it may be useful just to have helper macros.

```
define if\_def(\#) \equiv @\{@\&\$IFDEF\#@\}

define if\_not\_def(\#) \equiv @\{@\&\$IFNDEF\#@\}

define else\_if\_def(\#) \equiv @\{@\&\$ELSEIFDEFINED(\#)@\}

define else\_def \equiv @\{@\&\$ENDIF@\}

define end_if \equiv @\{@\&\$ENDIF@\}

define end\_if \equiv endif

format if\_def \equiv if

format if\_not\_def \equiv if

format else\_if\_def \equiv else

format else\_def \equiv else

format end_if \equiv end

format end_if \equiv end
```

11. Toggling IO Checking. Another compiler directive enables and disables IO checking

```
define disable\_io\_checking \equiv \mathbb{Q}\{0\&\$I-\mathbb{Q}\}
define enable\_io\_checking \equiv \mathbb{Q}\{0\&\$I+\mathbb{Q}\}
define without\_io\_checking(\#) \equiv disable\_io\_checking; \#; enable\_io\_checking
```

- 12. References. I have inline citations to the literature, but there's some references worth explicitly drawing the reader's attention to (which may or may not make it to an inline citation):
- (1) Andrzej Trybulec, "Some Features of the Mizar Language", ESPRIT Workshop, Torino, 1993. Eprint: mizar.uwb.edu.pl/project/trybulec93.pdf — §4 discusses grammatical aspects of Mizar
- (2) Freek Wiedijk, "Mizar's Soft Type System". In K. Schneider and J. Brandt, eds., *Theorem Proving in Higher Order Logics*. TPHOLs 2007, Springer, doi:10.1007/978-3-540-74591-4_28 (Eprint pdf).
- (3) Adam Grabowski, Artur Korniłowicz, and Adam Naumowicz's "Mizar in a Nutshell" (doi:10.6092/issn.1972-5787/1980)
- (4) Artur Korniłowicz's "Registrations vs Redefinitions in Mizar" (in A. Kohlhase, P. Libbrecht, BR. Miller, A. Naumowicz, W. Neuper, P. Quaresma, F. Wm. Tompa, M. Suda (eds) *Joint Proc. FM4M, MathUI, and ThEdu*, 2016, pp.17–20, ceur-ws.org/Vol-1785/F5.pdf)
- (5) Artur Korniłowicz's "On rewriting rules in Mizar" (*J. Autom. Reason.* **50** no.2 (2013) 203–210, doi:10.1007/s10817-012-9261-6)
- (6) Mario Carneiro, "Reimplementing Mizar in Rust". Eprint arXiv:2304.08391, see especially the first two sections for an overview of Mizar's workflow. (The code is available at github.com/digama0/mizarrs.)

I should also credit Wayne Sewell's Weaving a Program: Literate Programming in Web (Van Nostrand Reinhold Computer, 1989) for discussing how to take a pre-existing PASCAL program and turn it into a WEB. Or, depending on the quality of writing in this literate program, it's all his fault.

4 MIZAR'S WORKFLOW Mizar Parser §13

Section 0.1. MIZAR'S WORKFLOW

13. This section will give a brief overview of what Mizar "does" when we run it. The analogy to bear in mind is with a batch compiler: there's parsing, some intermediate steps, then emits some output.

Just to give some rough estimates of where Mizar spends most of its time, there are four phases Mizar reports when checking an article:

- (1) Parser (transforms input into an abstract syntax tree, writes it to an XML file);
- (2) MSM (transforms the abstract syntax tree into an explicitly typed intermediate representation) base/first_identification.pas, the MSMAnalyzer procedure; this will require transcribing kernel/limits.pas (which is mostly just a bunch of constant parameters);
- (3) Analyzer (performs type checking, tracks the goals, and other miscellaneous jobs) the *Analyze* procedure in kernel/analyzer.pas; this requires transcribing kernel code (lexicon.pas, inout.pas, iocorrel.pas, correl.pas, generato.pas, builtin.pas, justhan.pas, enums.pas, formats.pas, identify.pas) and base code (xmldict.pas), approximately 19590 lines (16764 lines of code, the rest is whitespace and comments)
- (4) Checker (performs the proof checking for validity) the *InferenceChecker* procedure in kernel/checker.pas. This requires transcribing kernel files (checker.pas prechecker.pas equalizer.pas unifier.pas justhan.pas), approximately 8191 lines of code.

Using numbers Mario Carneiro reported in his github repository, roughly 14/15 of Mizar's runtime (as measured in CPU time) is spent on the Analyzer and Checker phases (among which, Mizar spends about 5 times longer in the Checker phase than the Analyzer phase). Parsing and MSM transforms the input into an intermediate representation used in the latter two phases. Mizar spends about 1/15 of its time here.

- 14. Accommodator. This will produce, among other outputs, the ".dct" file (and its XML counterpart, the ".dcx" file). The ".dct" file contains all the identifiers imported from other articles and reserved keywords for Mizar. The Tokeniser needs it to properly tokenise an article.
- 15. Parsing phase. We can look at kernel/verfinit.pas to find the parsing phase of the Mizar program is handled by the following lines of code:

```
InitPass(`Parser_u'); FileExam(EnvFileName + `.dct`);
InitScanning(MizFileName + ArticleExt, EnvFileName); InitWSMizarArticle; Parse;
gFormatsColl.StoreFormats(EnvFileName + `.frx`); gFormatsColl.Done; FinishScanning;
Write_WSMizArticle(gWSTextProper, EnvFileName + `.wsx`);
```

Our goal is to examine these functions, and understand what is going on. We know Parse is defined in base/parse.pas, it populates the gWSTextProper global variable using base/parseraddition.pas, and $Write_WSMizArticle$ is defined in base/wsmarticle.pas. The Parse function continuously invokes ReadToken (§726).

This phase will be responsible for generating a ".frx" (formats XML) and a ".wsx" (weakly strict Mizar XML) file.

Section 0.2. LOG OF TODOS, BUGS, IMPROVEMENTS

16. I have a number of observations from transcribing Mizar into WEB. They're the last thing I have included in the introductory chapter.

17. Possible improvements.

- (1) In quicksort, picking the pivot is done by $P \leftarrow (Low + High)/2$, but it should be done by $P \leftarrow Low + ((High Low)/2)$ to avoid overflow.
- (2) Actually, quicksort should delegate work to a different sorting algorithm when there is less than 10 items in the list. Sedgewick pointed this out in his PhD thesis. (If quicksort were a culprit for slowness, we could even hardcode sort networks for small lists.)
- (3) We should also determine the pivot by looking at the median value of P = (3 * Low + High)/4, $P2 \leftarrow (Low + High)/2$, $P3 \leftarrow (Low + 3 * High)/4$. This will improve the performance of quicksort.
- (4) In §152, GCD could be optimized to avoid calculating Mul(i,i) in every loop iteration.
- (5) In §326, MStringList. ObjectOf has duplicate code.
- (6) It seems that parsing Mizar text, emitting XML, and parsing XML seem to contain a lot of code which could be autogenerated from a grammar (a hypothetical ".ebnf" file). This would avoid duplicate work.

- 18. Possible bugs. I have been working through the source code with the mindset of, "How can I possibly break this?" This has led me to identify a number of situations where things can "go badly". But they are not all bugs (some are impossible to occur).
- (1) In §168, RationalGT is either misnamed (should be RationalGEQ) or implemented incorrectly (we should have RationalGT(a,b) = RationalLT(b,a) but do not)
- (2) In §171 IsRationalGT is misnamed (should be IsRationalGEQ)
- (3) In §272, MSortedExtList.FindInterval appears to assume that MSortedExtList.Find returns the left-most index.
- (4) In §293, MSortedCollection. Search may not return the correct index when there are duplicates. This is not terrible, since IndexOf corrects for this possibility.
- (5) In §527, I think TXTStreamObj.Done needs to close the associated file.
- (6) In §563, TSymbol.Init expects an fInfinitive argument, but does not use it shouldn't it initialize $Infinite \leftarrow fInfinitive$?
- (7) In §501, escaped quotation marks are not properly handled.
- (8) For StreamObj (§520), the constructors and destructors are not virtual which would impact XMLOut-StreamObj (§533) well, we just do duplicate work in XMLOutStreamObj's constructors and destructors.
- (9) Shouldn't TokensCollection.InitTokens (§594) invoke the inherited constructor?
- (10) Shouldn't *MTokenObj.Init* (§602) invoke inherited constructors? At least to insulate itself from changes to any of its parent (or grandparent) classes?
- (11) The constructor OutWSMizFileObj.OpenFileWithXSL (§1015) expects the XML-stylesheet located at "file://'+\$MIZFILES+'/wsmiz.xml", but that file is not present in Mizar.
- (12) In extItemObj.FinishFunctorPattern (§1323), the default case does not add a new format to the gFormatsColl dictionary.
- (13) In CreateArgs (§1393) in parseraddition.pas, when $aBase \leq 0$, this will set TermNbr to a negative number.
- (14) In the Subexpression class, there is duplicate code (§1397) the *CompleteAttributeArguments* and *FinishAttributeArguments* are identical, but only the latter is consistent with the naming conventions for the parser. Or (probably more likely) I am misunderstanding the naming conventions?
- (15) In CompleteArgument (§1518), we should also test that fParenthCnt is positive, shouldn't we?
- (16) The CreateSubexpression method (§1491), for extended expression objects, may result in a memory leak when $gSubexpPtr \neq nil$ that is to say, if KillSubexpression has not been invoked prior to CreateSubexpression.
- (17) Misnamed variable: gIdenifyEqLociList should be gIdentifyEqLociList (i.e., "idenify" should be "identify" with a 't'). (This typo has been corrected in the literate presentation of the code.)
- (18) As discussed in (§1286), there is a mismatch between the documentation and the parser when it comes to parsing loci declarations in a definition block. The syntax.txt file is more restrictive than the parser, and should be updated to reflect the parser.
- (19) The qSuchThat global variable is never used anywhere (§1292)
- (20) In ATTSubexpression (§1641), in the **else** block when the conditional **if** $lAttrExp \lor (aExpKind = exAdjectiveCluster)$ is executed, aExpKind = exAdjectiveCluster is never true (so there's no need for it).

- **19. To do list.** There are some things I should revisit, revise, and edit specifically about this running commentary (*not* the Mizar source code).
- (1) [Missing transcription] I skipped over transcribing the *ItemName* and other constants from wsmarticle.pas, which I should probably include.
- (2) [Revise] The XML schema should use the doc/mizar/xml/Mizar.rnc schema snippets.
- (3) [Revise] Make an introduction to dynamic arrays as a data structure, just to standardize the terminology used. (Make sure I stick to the standardized terminology!) Including pictures may help...
- (4) [Revise] Review quicksort. I should prove that it works, too. (Has this been done in Mizar? exchsort seems to be the closest match.)
- (5) [Improve] Give a "big picture" summary of the architecture. For example, the most interesting routine in parsing Mizar, well, it's all handled in *MTokeniser.SliceIt* (§612).
- (6) [Linting] Standardize the names of basic data types. PASCAL accepts *integer* as synonymous with *Integer*, but they give different index entries.
- (7) [Cosmetics] Check the typography is correct for the code
- (8) [Cosmetics] Create more WEB macros for conditional compilation
- (9) [Cosmetics] Would it help to include more UML class diagrams?
- (10) [Improve] It may be useful to use UML State diagrams to explain the parser or it may be a huge distraction?
- **20.** Formatting types. This is still a finicky aspect of WEB. Strings are a type in Free PASCAL, like *Boolean*.

Looking at Wirth's book, he typesets a type in *italics* and lowercase — so we have *boolean* and not **boolean** or *Boolean* (or **Boolean** or boolean or...). Knuth's "TEX: the program" follows this convention (using *integer*, *boolean*, *char*, etc.).

21. Using Twill (or not). Knuth invented Twill as a "hack" atop WEAVE to include "mini-indices" every couple pages. The problem I have with Twill is that it does not adequately index local variables (in the sense that: Knuth's TEX is a giant monolithic program, and any var appearing in it is almost certainly a global variable — hence it makes sense to index *all* variables, since they are almost certainly global).

I want to use Twill, but it is designed specifically for Knuth. Consequently it is not terribly useful for our purposes. We would have to tailor it quite heavily, and I don't have the energy or patience to do so.

22. Caution: Knuth takes advantage of WEB to use snake_case when naming things instead of Pascal's idiomatic PascalCase. This probably greatly improves the readability of the code. We should probably think hard about using it.

When WEAVE extracts the PASCAL code, it will remove all underscores from the identifiers and capitalize all letters. So instead of "screaming_run_on_case" (which appears in the PDF), we will instead obtain "SCREAMINGRUNONCASE", which...yeah, that's a hot mess.

8 MIZAR ENVIRONMENT Mizar Parser $\S23$

File 1

end:

See also sections 28, 31, 35, and 46. This code is used in section 23.

Mizar environment

```
23. We will need some basic library of functions. For example, trimming whitespace from a String.
\langle \text{ mizenv.pas } 23 \rangle \equiv
  ⟨GNU License 4⟩
unit mizenv;
  interface
     (interface for mizenv.pas 24)
  implementation
     \langle Modules used by mizenv.pas 25 \rangle
     (implementation of mizenv.pas 26)
  end.
      The interface contains all the variables for the unit, and the public facing functions and procedures.
\langle \text{ interface for mizenv.pas } 24 \rangle \equiv
var MizFileName, EnvFileName, ArticleName, ArticleID, ArticleExt: String;
procedure SetStringLength(var aString : String; aLength : integer);
See also sections 27, 30, and 34.
This code is used in section 23.
     The implementation begins with various "uses".
\langle Modules used by mizenv.pas 25 \rangle \equiv
       { compiler dependent imports }
  if_{-}def (DELPHI) IOUtils, SysUtils, windows, endif
  if_def (FPC) dos, SysUtils, endif
  mconsole; { the only Mizar module mizenv.pas uses }
This code is used in section 23.
    As far as setting the String length, this is a straightforward implementation. When the desired aLength
is less than the actual length of aString, simply delete all characters after aLength.
  Otherwise, aString has fewer characters than desired, so we pad it on the right with however many spaces
until the String is as long as aLength.
\langle \text{ implementation of mizenv.pas } 26 \rangle \equiv
procedure SetStringLength(var aString : String; aLength : integer);
  var I, L: integer;
  begin L \leftarrow length(aString);
  if aLength \leq L then Delete(aString, aLength + 1, L - aLength)
  else for I \leftarrow 1 to aLength - L do aString \leftarrow aString + ' ' ';
```

§27 Mizar Parser MIZAR ENVIRONMENT 9

27. We have publicly available functions trimming whitespace from functions.

```
\langle \text{interface for mizenv.pas } 24 \rangle + \equiv  function TrimStringLeft(aString : String): String; function TrimStringRight(aString : String): String; function TrimString(\text{const } aString : String): String;
```

28. Trimming the left String will repeatedly delete any leading whitespace, until the String is empty or has no leading whitespace.

Similarly, trimming the right String will repeatedly delete the *last* character until it is no longer whitespace (or until the String becomes empty).

```
 \begin{split} &\langle \text{implementation of mizenv.pas } 26 \rangle + \equiv \\ & \textbf{function } \textit{TrimStringLeft}(aString: String) \text{: } \textit{String}; \\ & \textbf{begin while } (length(aString) > 0) \wedge (aString[1] = `\_`) \textbf{ do } \textit{Delete}(aString, 1, 1); \\ & \textit{TrimStringLeft} \leftarrow aString; \\ & \textbf{end}; \\ & \textbf{function } \textit{TrimStringRight}(aString: String) \text{: } \textit{String}; \\ & \textbf{begin while } (length(aString) > 0) \wedge (aString[length(aString)] = `\_`) \textbf{ do } \\ & \textit{Delete}(aString, length(aString), 1); \\ & \textit{TrimStringRight} \leftarrow aString; \\ & \textbf{end}; \\ \end{split}
```

29. Trimming a String amounts to trimming it on the left and right.

```
function TrimString(\mathbf{const}\ aString:\ String):\ String;
begin TrimString \leftarrow TrimStringRight(TrimStringLeft(aString));
end;
```

30. We have a few more String manipulation functions for changing case, and turning an integer into a String.

```
 \begin{array}{l} \langle \, {\rm interface \,\, for \,\, mizenv.pas \,\, 24} \, \rangle \, + \\ \\ {\rm function \,\,} UpperCase({\bf const \,\,} aStr: \,\, String) \colon \, String; \\ {\rm function \,\,} MizLoCase(aChar: \, char) \colon \, char; \\ {\rm function \,\,} LowerCase({\bf const \,\,} aStr: \,\, String) \colon \, String; \\ {\rm function \,\,} IntToStr(aInt: integer) \colon \, String; \end{array}
```

31. Now, uppercase letters.

```
\langle \text{ implementation of mizenv.pas } 26 \rangle + \equiv function UpperCase(\mathbf{const}\ aStr:\ String)\colon String; var k\colon integer;\ \{\text{ index ranging over } aStr\} lStr:\ String;\ \{\text{ the uppercased String being built and returned}\} begin lStr\leftarrow aStr; for k\leftarrow 1 to length(aStr) do lStr[k]\leftarrow UpCase(aStr[k]); UpperCase\leftarrow lStr; end:
```

10 MIZAR ENVIRONMENT Mizar Parser $\S32$

Lowercasing a String can be done by iteratively replacing each character with its lowercase version. This "lowercase a single character" function is precisely *MizLoCase*. **function** *MizLoCase(aChar:char): char;* begin if $aChar \in [`A`...`Z']$ then $MizLoCase \leftarrow Chr(Ord(`a`) + Ord(aChar) - Ord(`A`))$ else $MizLoCase \leftarrow aChar$; end: function LowerCase(const aStr: String): String; $var i: integer; \{index ranging over aStr's length\}$ lStr: String; { result being built up } **begin** $lStr \leftarrow aStr;$ for $i \leftarrow 1$ to length(aStr) do $lStr[i] \leftarrow MizLoCase(aStr[i]);$ $LowerCase \leftarrow lStr;$ end: **33.** We also want to convert an integer to a String. **function** IntToStr(aInt : integer): String; var lStr: String; **begin** Str(aInt, lStr); $IntToStr \leftarrow lStr$; end; **34.** File name manipulation. We will want to test if a file exists, or split a path (represented as a String) into a directory and a filename. $\langle \text{ interface for mizenv.pas } 24 \rangle + \equiv$ **function** MFileExists(**const** aFileName: String): boolean; **procedure** EraseFile(**const** aFileName: String); **procedure** RenameFile(const aName1, aName2: String); **function** GetFileTime(aFileName : String): Longint; **procedure** SplitFileName (**const** aFileName: String; **var** aDir, aName, aExt: String); **function** TruncDir(**const** aFileName: String): String: function TruncExt(const aFileName: String): String; **function** ExtractFileDir(**const** aFileName: String): String; function ExtractFileName (const aFileName: String): String; **function** ExtractFileExt(**const** aFileName: String): String; function ChangeFileExt(const aFileName, aFileExt: String): String; Testing if a file exists uses the Free Pascal's primitive FileExists function. Similarly, EraseFile is just relying on Free Pascal's SysUtils.DeleteFile function. $\langle \text{ implementation of mizenv.pas } 26 \rangle + \equiv$ **function** MFileExists(**const** aFileName: String): boolean; **begin** $MFileExists \leftarrow FileExists(aFileName)$; **end**; **procedure** EraseFile(**const** aFileName: String); **begin** SysUtils.DeleteFile(aFileName); **end**; We will destructively rename a file. If a file with the name already exists, we delete it. **procedure** RenameFile(**const** aName1, aName2: String); **begin if** MFileExists(aName1) **then** EraseFile(aName2);

SysUtils.RenameFile(aName1, aName2);

end;

ξ37 Mizar Parser MIZAR ENVIRONMENT 11

Again, relying on Free Pascal's FileAge function. **function** GetFileTime(aFileName : String): Longint; **begin** $GetFileTime \leftarrow FileAge(aFileName)$; **end**; **38.** Split a file name into components, namely (1) the directory, (2) the file name, (3) its extension. For example, /path/to/my/file.exe will be split into /path/to/my/, file, and exe. This implementation depends on the compiler used (Delphi or Free Pascal). **procedure** SplitFileName (const aFileName: String; { input } var aDir, aName, aExt: String); { output } begin $if_{-}def(FPC)$ $aDir \leftarrow SysUtils.ExtractFilePath(aFileName);$ $aName \leftarrow SysUtils.ExtractFileName(aFileName);$ $aExt \leftarrow SysUtils.ExtractFileExt(aFileName);$ endif **if_def** (DELPHI) $aDir \leftarrow TPath.GetDirectoryName(aFileName);$ $aName \leftarrow TPath.GetFileName(aFileName);$ $aExt \leftarrow TPath.GetExtension(aFileName);$ endif end: "Truncating a directory" means "throw away the directory part of the path" so we end up with just a filename and the file extension. function TruncDir(const aFileName: String): String; var Dir, lName, Ext: String; **begin** $SplitFileName(aFileName, Dir, lName, Ext); TruncDir <math>\leftarrow lName + Ext;$ end: 40. "Truncating the extension" means throwing away the extension part of a path. function TruncExt(const aFileName: String): String; var Dir, lName, Ext: String; **begin** $SplitFileName(aFileName, Dir, lName, Ext); TruncExt \leftarrow Dir + lName;$ end; **41.** Extracting the file directory will return *just* the directory part of a path. **function** ExtractFileDir(**const** aFileName: String): String; var Dir, lName, Ext: String; **begin** $SplitFileName(aFileName, Dir, lName, Ext); ExtractFileDir <math>\leftarrow Dir;$ end: 42. Extracting the file name will throw away both the directory and extension. **function** ExtractFileName(**const** aFileName: String): String; var Dir, lName, Ext: String; **begin** $SplitFileName(aFileName, Dir, lName, Ext); ExtractFileName <math>\leftarrow lName;$ end: function ExtractFileExt(const aFileName: String): String; var Dir, lName, Ext: String;

begin $SplitFileName(aFileName, Dir, lName, Ext); ExtractFileExt <math>\leftarrow Ext;$

end;

12 MIZAR ENVIRONMENT Mizar Parser $\S43$

```
Changing a file name's extension. See:
  https://www.freepascal.org/docs-html/rtl/sysutils/changefileext.html
function ChangeFileExt(const aFileName, aFileExt: String): String;
  begin ChangeFileExt \leftarrow SysUtils.ChangeFileExt(aFileName, aFileExt); end;
44. Getting an environmental variable.
function GetEnvStr(aEnvName : String): String;
    if_{-}def(FPC)
       begin GetEnvStr \leftarrow GetEnv(aEnvname); end;
    if_def (DELPHI) (Get environment variable, Delphi-compatible mode 45)
    endif
     The Delphi-compatible version of obtaining an environment variable is rather involved, so let us study
it in silent meditation.
\langle Get environment variable, Delphi-compatible mode 45\rangle \equiv
const cchBuffer = 255;
var lName, lpszTempPath: array [0...cchBuffer] of char;
  i: integer; lStr: String;
  begin lStr \leftarrow ::
  for i \leftarrow 1 to length(aEnvname) do lName[i-1] \leftarrow aEnvname[i];
  lName[length(aEnvname)] \leftarrow #0;
  if GetEnvironmentVariable(lName, lpszTempPath, cchBuffer) > 0 then
    begin for i \leftarrow 0 to cchBuffer do
       begin if lpszTempPath[i] = \#0 then break;
       lStr \leftarrow lStr + lpszTempPath[i];
      end:
    end;
  GetEnvStr \leftarrow lStr;
    { restored for DELPHI4 compatibility ;-( begin GetEnvStr:=GetEnvironmentVariable(aEnvname);
      end; }
This code is used in section 44.
     The DrawMessage and DrawIOResult are procedures in mconsole.pas (see §74, and §75).
\langle \text{ implementation of mizenv.pas } 26 \rangle + \equiv
procedure FileExam(const aFileName: String);
  var
  Source: file;
  I: byte;
    begin if aFileName =  then
       begin DrawMessage('Can' 't⊔open⊔' 'L'.miz⊔' ', ''); halt(1);
      end;
    FileMode \leftarrow 0; \ Assign(Source, aFileName); \ without\_io\_checking(Reset(Source)); \ I \leftarrow ioresult;
    if I \neq 0 then DrawIOResult(aFileName, I);
    close(Source); FileMode \leftarrow 2;
    end;
```

 $\S47$ Mizar Parser MIZAR ENVIRONMENT 13

47. We test if a file exists with the *EnvFileExam* procedure. It will notify the user if the file does not exist, otherwise it is silent.

```
Again, DrawMessage comes from mconsole.pas (§74).
procedure EnvFileExam(const aFileExt: String);
  begin if \neg MFileExists(EnvFileName + aFileExt) then
     \mathbf{begin} \ \mathit{DrawMessage}(\ \texttt{`Can'`t}_{\square} \mathsf{open}_{\square} \ \texttt{`}_{\square}' + \mathit{EnvFileName} + \mathit{aFileExt} + \texttt{`}_{\square}'', \texttt{`}'); \ \mathit{Halt}(1);
     end;
  end;
     The ParamCount is PASCAL's way of counting the command-line parameters passed to the program.
procedure GetFileName(ParamNr: byte; DefaultExt: String; var aFileName: String);
  var lFileExt: String;
  begin if ParamNr \leq ParamCount then
     begin aFileName \leftarrow ParamStr(ParamNr); lFileExt \leftarrow ExtractFileExt(aFileName);
     if lFileExt =  then aFileName \leftarrow ChangeFileExt(aFileName, DefaultExt);
     exit
     end;
  aFileName \leftarrow ::
  end;
     This procedure will take the Nr command line argument. If it lacks a file extension, then it will append
the DefaultExt to it. At the end, this will populate aFileName and aFileExt based on the command line.
  The ParamStr(Nr) returns the Nr^{th} parameter as a String (it's a PASCAL primitive).
procedure GetFileExtName(Nr: byte; DefaultExt: String; var aFileName: String; var aFileExt: String);
  begin if Nr \leq ParamCount then
     begin aFileName \leftarrow ParamStr(Nr); aFileExt \leftarrow ExtractFileExt(aFileName);
     if aFileExt =  ``then aFileExt \leftarrow DefaultExt
     else aFileName \leftarrow ChangeFileExt(aFileName, ``);
     exit
     end;
```

50. We need to find the first command-line argument which resembles a Mizar article name. Note that Mizar articles have several files associated with it (the article's contents in a .miz file, the vocabulary in a .voc file, and XML related intermediate representation in .xml files, as well as .evl files).

 $aFileName \leftarrow ``; aFileExt \leftarrow ``;$

end;

Command line flags prefixed with a dash ("-") will not be interpreted as the name of a Mizar article.

```
procedure GetMizFileName (aFileExt: String);

var i: integer;

begin MizFileName \leftarrow ```; ArticleName \leftarrow ```; ArticleExt \leftarrow ```; EnvFileName \leftarrow ```;

for i \leftarrow 1 to ParamCount do

if ParamStr(i)[1] \neq `-` then

begin MizFileName \leftarrow ParamStr(i); GetFileExtName(i, aFileExt, MizFileName, ArticleExt);

ArticleName \leftarrow ExtractFileName(MizFileName); ArticleID \leftarrow UpperCase(ArticleName);

if \neg IsMMLIdentifier(ArticleName) then

begin DrawMessage(`Onlyuletters,unumbersuanduualloweduinuMizarufileunames`, ```);

halt(1);
end;
EnvFileName \leftarrow MizFileName; exit;
end;
end;
```

14 MIZAR ENVIRONMENT Mizar Parser $\S51$

51. We will provide a standard way to refer to the article. procedure GetArticleName; begin GetMizFileName('.miz'); end: **52.** The *EnvFileName* is populated here provided *MizFileName* is blank. procedure GetEnvironName; **var** i, c: integer; **begin if** *MizFileName* = ``then *GetArticleName*; $EnvFileName \leftarrow MizFileName; c \leftarrow 0;$ for $i \leftarrow 1$ to ParamCount do if $(ParamStr(i)[1] \neq `-`)$ then **begin** inc(c); if c = 2 then $EnvFileName \leftarrow ParamStr(i)$; end; end; 53. The valid characters which can appear in a Mizar article name (an "MML Identifier") are uppercase Latin letters (A-Z), lowercase Latin letters (a-z), decimal digits (0-9), and underscores (_). **function** *IsMMLIdentifier*(**const** *aID*: *String*): *boolean*; const Allowed: array [chr(0) ... chr(255)] of **var** *i*: *integer*; **begin for** $i \leftarrow 1$ **to** length(aID) **do**

if Allowed[aID[i]] = 0 then

 $IsMMLIdentifier \leftarrow true;$

end;

begin $IsMMLIdentifier \leftarrow false; exit; end;$

§54 Mizar Parser MIZAR CONSOLE 15

File 2

This code is used in section 57.

Mizar Console

```
54. The Mizar Console unit is used for interacting with the command line.
\langle \text{ mconsole.pas } 54 \rangle \equiv
  ⟨GNU License 4⟩
unit mconsole;
  interface (Interface for moonsole.pas 56)
  implementation
     \langle \, \text{Import units for mconsole.pas } 55 \, \rangle
     (Implementation for mconsole.pas 65)
  end
55. We import two modules, pcmizver (which we have yet to see), and mizenv (which we have already
\langle \text{Import units for mconsole.pas } 55 \rangle \equiv
uses pcmizver, mizenv;
This code is used in section 54.
     The interface contains the publicly available functions, as well as some specific variables for the state
of the analyzer, etc.
\langle Interface for mconsole.pas 56\rangle \equiv
  (Report results to command line 57)
   (Constants for common error messages reported to console 60)
   (Interface for accommodator command line options 61)
   \langle \text{Interface for } MakeEnv \text{ command line options } 62 \rangle
   (Interface for transfer-specific command line options 63)
  \langle\, {\rm Interface} \ {\rm for} \ {\rm other} \ {\rm command} \ {\rm line} \ {\rm options} \ {\rm 64} \, \rangle
This code is used in section 54.
57. \langle Report results to command line 57 \rangle \equiv
procedure InitDisplayLine(const aComment: String);
procedure NoDisplayLine(fLine, fErrNbr: integer);
  ⟨ DisplayLine global constant 58 ⟩
procedure DrawMizarScreen(const aApplicationName: String);
procedure DrawArticleName(const fName: String);
procedure DrawStr(const aStr: String);
procedure FinishDrawing;
See also section 59.
This code is used in section 56.
58. \langle DisplayLine global constant 58 \rangle \equiv
const DisplayLine: procedure (fLine, fErrNbr : integer) = NoDisplayLine;
```

16 MIZAR CONSOLE Mizar Parser §59

```
59.
     Common routines for "drawing" output to the console.
\langle Report results to command line 57 \rangle + \equiv
procedure EmptyParameterList;
procedure Noise;
procedure DrawPass(const aName: String);
procedure DrawTime(const aTime: String);
procedure DrawVerifierExit(const aTime: String);
procedure DrawMessage(const Msg1, Msg2: String);
procedure BugInProcessor;
procedure DrawIOResult(const FileName: String; I: byte);
procedure DrawErrorsMsg(aErrorNbr : integer);
procedure DisplayLineInCurPos(fLine, fErrNbr: integer);
    We also have a constant for error messages commonly encountered.
\langle Constants for common error messages reported to console 60 \rangle \equiv
const ErrMsg: array [1..6] of String[20] =
  (--,
  'File not found',
  'Path, not, found',
  'Too⊔many⊔open⊔files',
  'Disk_read_error',
  'Disk_write_error');
This code is used in section 56.
61. Now, we have accommodator specific options.
\langle \text{Interface for accommodator command line options } 61 \rangle \equiv
     { Accommodator specific options: }
\mathbf{var}\ Insert Hidden Files\ ,\ Vocabularies\ Processing\ ,\ Formats\ Processing\ ,\ Notations\ Processing\ ,\ Signature\ Processing\ ,
       ConstructorsProcessing, ClustersProcessing, IdentifyProcessing, ReductionProcessing,
       Properties Processing, Definitions Processing, Equalities Processing, Expansions Processing,
       Theorems Processing, Schemes Processing, Theorem Lists Processing, Scheme Lists Processing:\ boolean;
procedure InitAccOptions;
procedure GetAccOptions;
This code is used in section 56.
     \langle \text{Interface for } MakeEnv \text{ command line options } 62 \rangle \equiv
     { MakeEnv specific options: }
var Accomodation: boolean = false; NewAccom: boolean = false;
procedure GetMEOptions;
This code is used in section 56.
63.
      \langle Interface for transfer-specific command line options 63\rangle \equiv
    { Transfer specific options: }
var PublicLibr: boolean;
procedure GetTransfOptions;
This code is used in section 56.
```

§64 Mizar Parser MIZAR CONSOLE 17

```
64. ⟨Interface for other command line options 64⟩ ≡
{Other options:}
var CtrlCPressed: boolean = false; LongLines: boolean = false; QuietMode: boolean = false;
StopOnError: boolean = false; FinishingPass: boolean = false; ParserOnly: boolean = false;
AnalyzerOnly: boolean = false; CheckerOnly: boolean = false; SwitchOffUnifier: boolean = false;
AxiomsAllowed: boolean = false;
procedure GetOptions;
This code is used in section 56.
65. The implementation begins by initializing the Accommodator specific options.
⟨Implementation for mconsole.pas 65⟩ ≡
```

 $\begin{array}{l} \textbf{procedure} \ InitAccOptions; \\ \textbf{begin} \ InsertHiddenFiles \leftarrow true; \ VocabulariesProcessing \leftarrow true; \ FormatsProcessing \leftarrow true; \\ NotationsProcessing \leftarrow true; \ SignatureProcessing \leftarrow true; \ ConstructorsProcessing \leftarrow true; \\ ClustersProcessing \leftarrow true; \ IdentifyProcessing \leftarrow true; \ ReductionProcessing \leftarrow true; \\ PropertiesProcessing \leftarrow true; \ DefinitionsProcessing \leftarrow true; \ EqualitiesProcessing \leftarrow true; \\ ExpansionsProcessing \leftarrow true; \ TheoremSProcessing \leftarrow true; \ SchemeSProcessing \leftarrow true; \\ TheoremListsProcessing \leftarrow false; \ SchemeListsProcessing \leftarrow false; \\ \textbf{end}; \end{array}$

This code is used in section 54.

66. Similarly, we want to be able to *reset* the configuration for the accommodator to the default (initial) values.

```
procedure ResetAccOptions;
```

```
 \begin{array}{l} \textbf{begin} \ \textit{InsertHiddenFiles} \leftarrow \textit{true}; \ \textit{VocabulariesProcessing} \leftarrow \textit{false}; \ \textit{FormatsProcessing} \leftarrow \textit{false}; \\ \textit{NotationsProcessing} \leftarrow \textit{false}; \ \textit{SignatureProcessing} \leftarrow \textit{false}; \ \textit{ConstructorsProcessing} \leftarrow \textit{false}; \\ \textit{ClustersProcessing} \leftarrow \textit{false}; \ \textit{IdentifyProcessing} \leftarrow \textit{false}; \ \textit{ReductionProcessing} \leftarrow \textit{false}; \\ \textit{PropertiesProcessing} \leftarrow \textit{false}; \ \textit{DefinitionsProcessing} \leftarrow \textit{false}; \ \textit{EqualitiesProcessing} \leftarrow \textit{false}; \\ \textit{ExpansionsProcessing} \leftarrow \textit{false}; \ \textit{TheoremSProcessing} \leftarrow \textit{false}; \\ \textit{TheoremListsProcessing} \leftarrow \textit{false}; \ \textit{SchemeListsProcessing} \leftarrow \textit{false}; \\ \textbf{end}; \\ \end{array}
```

18 MIZAR CONSOLE Mizar Parser $\S67$

67. Accommodator options. We will get options for the accommodator passed in from the command line. Broadly, these are:

- -v resets the accommodator options, and then toggles Vocabularies Processing to true
- -f, -p resets the accommodator options, and then toggles *VocabulariesProcessing* to true (so far like -v), and then toggles *FormatsProcessing* to true.
- -P resets the accommodator options, and then toggles *VocabulariesProcessing* to true (so far like -v), and then toggles *FormatsProcessing* to true (so far like -f and -p), then toggles *TheoremListsProcessing* and *SchemeListsProcessing* to both be true.
- -e will do everything -f does, and then toggles ConstructorsProcessing, SignatureProcessing, ClustersProcessing, and NotationsProcessing to all be true.
- -h will set *InsertHiddenFalse* to false (presumably preventing Mizar from loading the "hidden" article, i.e., the primitive notions of object, set, in, =, and inequality will not be loaded).
- -1 will toggle *LongLines* to true (allowing lines with more than 80 characters)
- -q will toggle *QuietMode* to true
- ullet -s will toggle StopOnError to true

Note this processes all command line options in order. So -e -v will produce the same results as -v alone.

```
procedure GetAccOptions;
  var i, j: integer;
  begin InitAccOptions;
  for j \leftarrow 1 to ParamCount do
     if ParamStr(j)[1] =  '-' then
       for i \leftarrow 2 to length(ParamStr(j)) do
          case ParamStr(j)[i] of
          `v`: \mathbf{begin} \ ResetAccOptions; \ VocabulariesProcessing \leftarrow true
          f', p': begin ResetAccOptions; VocabulariesProcessing \leftarrow true; FormatsProcessing \leftarrow true;
            end:
          'P': begin ResetAccOptions; VocabulariesProcessing \leftarrow true; FormatsProcessing \leftarrow true;
             TheoremListsProcessing \leftarrow true; SchemeListsProcessing \leftarrow true;
           'e': begin ResetAccOptions; VocabulariesProcessing \leftarrow true; FormatsProcessing \leftarrow true;
             ConstructorsProcessing \leftarrow true; SignatureProcessing \leftarrow true; ClustersProcessing \leftarrow true;
             NotationsProcessing \leftarrow true;
          'h': begin InsertHiddenFiles \leftarrow false; end;
           1: LongLines \leftarrow true;
           q: QuietMode \leftarrow true;
          s: StopOnError \leftarrow true;
          endcases;
  end:
```

§68 Mizar Parser MIZAR CONSOLE 19

Similarly, we have *MakeEnv* specific options parsed from the command line flags. procedure GetMEOptions; **var** i, j: integer; begin for $j \leftarrow 1$ to ParamCount do if ParamStr(j)[1] = `-` then for $i \leftarrow 2$ to length(ParamStr(j)) do case ParamStr(j)[i] of $in: NewAccom \leftarrow true;$ $`a`: Accomodation \leftarrow true;$ $`l`: LongLines \leftarrow true;$ $q: QuietMode \leftarrow true;$ $s: StopOnError \leftarrow true;$ endcases; end; The "other" options. Notably, there is a feature to allow axioms, which is completely undocumented (and probably for good reason!). The axioms must appear in ".axm" files. **procedure** GetOptions; **var** i, j: integer; begin for $j \leftarrow 1$ to ParamCount do if ParamStr(j)[1] = `-` then for $i \leftarrow 2$ to length(ParamStr(j)) do case ParamStr(j)[i] of $q: QuietMode \leftarrow true;$ $p: ParserOnly \leftarrow true;$ $`a`: AnalyzerOnly \leftarrow true;$ $c: CheckerOnly \leftarrow true;$ $1: LongLines \leftarrow true;$ $s: StopOnError \leftarrow true;$ $u': SwitchOffUnifier \leftarrow true;$ \mathbf{x} : $AxiomsAllowed \leftarrow true$; othercases break; endcases; if $ArticleExt = `.axm' then AxiomsAllowed \leftarrow true;$ end; **70.** Transfer specific options. **procedure** GetTransfOptions; var lOption: String; **begin** $PublicLibr \leftarrow false;$ if $ParamCount \geq 2$ then **begin** $lOption \leftarrow ParamStr(2)$; if $(logth(lOption) = 2) \land (lOption[1] \in [^{\prime}/^{\prime}, ^{\prime}-^{\prime}])$ then $PublicLibr \leftarrow UpCase(lOption[2]) = ^{\circ}P^{\prime};$

 \mathbf{end} ;

20 MIZAR CONSOLE Mizar Parser $\S71$

```
We have a number of functions useful for "drawing", i.e., reporting progress and results (and so on).
var gComment: String = ``;
  disable_io_checking;
procedure NoDisplayLine(fLine, fErrNbr: integer);
  begin end;
procedure InitDisplayLine(const aComment: String);
  begin qComment \leftarrow aComment; WriteLn; write(aComment); DisplayLine \leftarrow DisplayLineInCurPos
  end;
procedure DrawStr(const aStr: String);
  begin write(aStr) end;
procedure FinishDrawing;
  begin WriteLn;
  end:
procedure DrawTPass(const fPassName: String);
  begin write(fPassName) end;
procedure DrawMizarScreen(const aApplicationName: String);
  begin WriteLn(aApplicationName, `, , ', PCMizarVersionStr, `, ', (', PlatformNameStr, `)');
  WriteLn(Copyright);
  end;
procedure Noise;
  begin
  if_not_def (WIN32) write (\uparrow G \uparrow G \uparrow G \uparrow G);
  endif;
  end;
procedure EmptyParameterList;
  begin Noise; WriteLn; WriteLn('****⊔⊔Empty⊔Parameter⊔List⊔?⊔****'); halt(2);
  end;
72. More such procedures, reporting the article processed, the time, etc.
procedure DrawArticleName(const fName: String);
  begin WriteLn('Processing: __', fName); end;
procedure DrawPass(const aName: String);
  begin WriteLn; write(aName); end;
procedure DrawTime(const aTime: String);
  begin write(aTime); end;
procedure DrawVerifierExit(const aTime: String);
  begin WriteLn; WriteLn(`Time_lof_lmizaring:`, <math>aTime);
  end;
```

§73 Mizar Parser MIZAR CONSOLE 21

```
73.
procedure DisplayLineInCurPos(fLine, fErrNbr: integer);
  begin if (\neg CtrlCPressed) \land (\neg QuietMode) then
    begin write(\uparrow M + gComment + \vdash_{\sqcup} [\vdash, fLine : 4);
    if fErrNbr > 0 then write(`\_*`, fErrNbr);
    write(`]`);
    end:
  if FinishingPass then
    begin write(` [ `, fLine : 4);
    if fErrNbr > 0 then write(`` +`, fErrNbr);
    write(`]`);
    end;
  end;
74.
procedure DrawMessage(const Msg1, Msg2: String);
  var Lh: byte;
  begin Noise; WriteLn; write(****_{\bot}, Msg1); Lh \leftarrow length(Msg1);
  if length(Msg2) > Lh then Lh \leftarrow length(Msg2);
  if Lh > length(Msg1) then write(``Lh - length(Msg1));
  WriteLn(`\_****`);
  if Msg2 \neq  then
    begin write("****", Msg2");
    if Lh > length(Msg2) then write(` : Lh - length(Msg2));
    WriteLn(`\_****`);
    end;
  end;
75.
procedure BugInProcessor;
  begin DrawMessage('Internal_Error', ''); end;
procedure DrawIOResult(const FileName: String;
  I: byte);
  begin if I \in [2 ... 6, 12, 100] then
    begin if I = 12 then I \leftarrow 7
    else if I = 100 then I \leftarrow 8;
    DrawMessage(ErrMsg[I], Can^{t_{\sqcup}}open_{\sqcup}^{-t_{\sqcup}} + FileName + ^{t_{\sqcup}})
  else DrawMessage('Can''tuopenu''' + FileName + 'u''', '');
  halt(1);
  end;
procedure DrawErrorsMSg(aErrorNbr: integer);
  begin if aErrorNbr > 0 then
    begin WriteLn;
    if aErrorNbr = 1 then WriteLn(`****_{\sqcup}1_{\sqcup}error_{\sqcup}detected`)
    else WriteLn( **** → , aErrorNbr, ~ uerrors udetected ~);
    end;
  end;
```

22 PC MIZAR VERSION Mizar Parser §77

File 3

PC Mizar Version

```
77. This is used to track the version of Mizar.
\langle \text{ pcmizver.pas } 77 \rangle \equiv
  (GNU License 4)
unit pcmizver;
  interface
  \mathbf{const}\ \mathit{PCMizarReleaseNbr} = 8;
     PCMizarVersionNbr = 1;
     PCMizarVariantNbr = 14;
     Current Year = 2025;
     Q{Q\&\$IFDEF\ WIN32Q}DirSeparator = `\`;
     @{0\&\$ELSE@}DirSeparator = '/';
     @{@&$ENDIF@}
78. There are only four functions provided by this module.
function PCMizarVersionStr: String;
function VersionStr: String;
function PlatformNameStr: String;
function Copyright: String;
79.
     Their implementation is relativiely straightforward: just print the appropriate constants to the screen.
implementation
function Copyright: String;
  var s: String;
  begin Str(CurrentYear, s);
  Copyright \leftarrow \texttt{`Copyright}_{\sqcup}(c)_{\sqcup}1990-\texttt{`}+s+\texttt{`}_{\sqcup}Association_{\sqcup}of_{\sqcup}Mizar_{\sqcup}Users\texttt{'};
  end;
80.
function VersionStr: String;
  \mathbf{var}\ lRel, lVer, lVar:\ String[2];\ lStr:\ String;
  \mathbf{begin} \ Str(PCMizarReleaseNbr, lRel); \ Str(PCMizarVersionNbr, lVer); \ Str(PCMizarVariantNbr, lVar);
  if length(lVar) = 1 then lVar \leftarrow `0' + lVar;
  @{@\&\$IFDEF\ VERALPHA@}lStr \leftarrow \text{`-alpha'};
  0{0\&\$ELSE0}lStr \leftarrow :;
  @{@&$ENDIF@}
  VersionStr \leftarrow lRel + `.` + lVer + `.` + lVar + lStr;
  end;
```

 $\S 81$ Mizar Parser PC MIZAR VERSION 23

81. There are a number of platforms supported, a surprisingly large number. If we were to support more platforms (other BSDs, BeOS, GNU Hurd, etc.), then we would need to update this function. To see what platforms are predefined for FreePascal, consult:

• https://wiki.freepascal.org/Platform_defines

Ostensibly, we could extend the platform name string to display "generic UNIX" (and even "generic BSD"), as well as "generic Windows".

```
function PlatformNameStr: String;
var lStr: String;
begin lStr \leftarrow ```;
if_def (WIN32)lStr \leftarrow lStr + `Win32`; end_if
if_def (LINUX)lStr \leftarrow lStr + `Linux`; end_if
if_def (SOLARIS)lStr \leftarrow lStr + `Solaris`; end_if
if_def (FREEBSD)lStr \leftarrow lStr + `FreeBSD`; end_if
if_def (DARWIN)lStr \leftarrow lStr + `Darwin`; end_if
if_def (FPC)lStr \leftarrow lStr + `/FPC`; end_if
if_def (DELPHI)lStr \leftarrow lStr + `/Delphi`; end_if
PlatformNameStr \leftarrow lStr;
end;
```

82. The last function in the pcmizver.pas file provides a String for the Mizar version.

```
 \begin{array}{ll} \textbf{function} \ \mathit{PCMizarVersionStr} \colon \mathit{String}; \\ \textbf{begin} \ \mathit{PCMizarVersionStr} \leftarrow \texttt{`Mizar} \sqcup \texttt{Ver} \cdot \sqcup \texttt{`} + \mathit{VersionStr}; \\ \textbf{end}; \\ \textbf{end} \ . \end{array}
```

24 MIZAR INTERNAL STATE Mizar Parser §83

83. As far as processing an article, Mizar works like a "batch compiler" and works in multiple "passes".

File 4

Mizar internal state

```
\langle \text{ mstate.pas } 83 \rangle \equiv
  (GNU License 4)
unit mstate;
  interface
    ⟨Interface for mstate.pas 84⟩
  implementation
  uses mizenv, pemizver, monitor, errhan, meonsole, mtime
  mdebug , info end_mdebug ;
  var PassTime: longint;
    ⟨Implementation for mstate.pas 85⟩
  end
84. \langle \text{Interface for mstate.pas } 84 \rangle \equiv
procedure InitPass(const aPassName: String);
procedure FinishPass;
procedure InitProcessing(const aProgName, aExt: String);
procedure ProcessingEnding;
This code is used in section 83.
     The implementation amounts to, well, these four functions. We have a couple "private" functions to
help us: MError and MizarExitProc.
\langle \text{Implementation for mstate.pas } 85 \rangle \equiv
procedure InitPass(const aPassName: String);
  begin CurPos.Line \leftarrow 1; CurPos.Col \leftarrow 1; InitDisplayLine(aPassName); TimeMark(PassTime);
  end;
procedure FinishPass;
  begin FinishingPass \leftarrow true;
  if QuietMode then DisplayLine(CurPos.Line, ErrorNbr);
  FinishingPass \leftarrow false; \ DrawTime(``\lumber' + ReportTime(PassTime));
  end;
procedure MError(Pos : Position; ErrNr : integer);
  begin WriteError(Pos, ErrNr); DisplayLine(CurPos.Line, ErrorNbr);
  end:
This code is used in section 83.
```

 $\S86$ Mizar Parser MIZAR INTERNAL STATE 25

```
We also have MizarExitProc as a private "helper" function.
86.
var _ExitProc: pointer;
procedure MizarExitProc;
  begin ExitProc \leftarrow \_ExitProc;
  disable_io_checking;
  if IOResult \neq 0 then;
  if ¬StopOnError then DisplayLine(CurPos.Line, ErrorNbr);
  PutError \leftarrow WriteError; DrawVerifierExit(ReportTime(gStartTime)); \{Halt(ErrorCode); \}
  enable_io_checking;
  end;
87.
procedure InitProcessing(const aProgName, aExt: String);
  begin DrawMizarScreen(aProgName);
  \label{eq:count} \textbf{if} \ \textit{ParamCount} < 1 \ \textbf{then} \ \textit{EmptyParameterList};
  GetArticleName; GetEnvironName; DrawArticleName (MizFileName + aExt); GetOptions;
  InitExitProc;\ FileExam(MizFileName + aExt);\ \_ExitProc \leftarrow ExitProc;\ ExitProc \leftarrow @MizarExitProc;
  PutError \leftarrow MError; OpenErrors(MizFileName);
  mdebug OpenInfoFile; end_mdebug
  end;
88. At the end, we should report the number of errors (if any were encountered).
procedure ProcessingEnding;
  begin if ErrorNbr > 0 then
    begin DrawErrorsMsq(ErrorNbr); FinishDrawing;
    end;
  end;
```

26 MONITOR Mizar Parser §89

File 5

Monitor

```
89.
\langle monitor.pas 89\rangle \equiv
  ⟨GNU License 4⟩
unit monitor;
  {\bf interface}
  procedure InitExitProc;
    implementation\\
     uses
       @{@\&\$IFDEFFPC}
          @{@&$IFNDEF WIN32@}
       baseunix,
          @{@&$ENDIF@}
       @\{@\&\$ENDIF\,@\}\\
       mizenv, errhan, mconsole
        @ \{ @ \& \$IFDEF\ WIN32\ @ \}\ , windows\ @ \{ @ \& \$ENDIF\ @ \} 
          mdebug , info end_mdebug
     \mathbf{var} \ \_ExitProc: \ pointer; \ \_IOResult: \ integer;
       \langle Implementation for monitor.pas 90\rangle
     end
```

 $\S90$ Mizar Parser MONITOR 27

```
There are a few private helper functions in this module.
\langle \text{Implementation for monitor.pas } 90 \rangle \equiv
procedure _Halt_(ErrorCode : word);
      begin \_IOResult \leftarrow IOResult; ErrorAddr \leftarrow nil;
      if ErrorCode > 1 then
            case ErrorCode of
            2...4: begin ErrImm(1000 + ErrorCode); DrawMessage(`I/O_{\square}error`, ErrMsg[ErrorCode]) end;
            5...6: begin ErrImm(1000 + ErrorCode); BugInProcessor end;
            12: begin ErrImm(1000 + ErrorCode); BugInProcessor end;
            97, 98, 99: begin ErrImm(RTErrorCode); \langle Handle runtime error cases for monitor.pas 91\rangle
                  end:
            100 .. 101: begin ErrImm(1000 + ErrorCode); DrawMessage('I/O⊔error', ErrMsq[ErrorCode − 95]);
                  end;
            102...106: begin ErrImm(1000 + ErrorCode); BugInProcessor end;
            150 ... 162: begin ErrImm(1000 + ErrorCode);
                  DrawMessage('I/O⊔error', 'Critical⊔disk⊔error');
                  end;
            200 . . 201: begin ErrImm(1000 + ErrorCode); BugInProcessor end;
            202: begin ErrImm(1000 + ErrorCode); DrawMessage(`Stack_loverflow_lerror', ``) end;
            203, 204: begin ErrImm(1000 + ErrorCode); DrawMessage(`Heaploverflow_error', ``) end;
            208: begin ErrImm(1000+ErrorCode); DrawMessage('Overlay manager not installed','') end;
            209: begin ErrImm(1000 + ErrorCode); DrawMessage(`Overlay_ifile_iread_ierror`, ``) end;
            210 ... 212: begin ErrImm(1000 + ErrorCode); BuqInProcessor end;
            213: begin ErrImm(1000 + ErrorCode); DrawMessage(`Collection_{\sqcup}Index_{\sqcup}out_{\sqcup}of_{\sqcup}range', ``) end;
            214: begin ErrImm(1000 + ErrorCode); DrawMessage(`Collection_overflow_error', `Collection_overflow_error', `Collection_overflow_e
           215: begin ErrImm(1000 + ErrorCode); DrawMessage(`Arithmetic\_overflow\_error`, `Constant Constant C
            216: begin ErrImm(1000 + ErrorCode); DrawMessage(General_Protection_fault, ) end;
            217: begin ErrImm(1000 + ErrorCode); DrawMessage(`Segmentation_lfault`, ``) end;
            218...254: begin ErrImm(1000 + ErrorCode); BugInProcessor end;
            255: ErrImm(1000 + ErrorCode);
            othercases begin ErrImm(ErrorCode);
                  if OverflowErroor then DrawMessage('Mizar_parameter_overflow_error', '`)
                 else BuqInProcessor
                  end;
            endcases:
      CloseErrors; ExitProc \leftarrow \_ExitProc;
      if (ErrorCode = 0) \land (ErrorNbr \neq 0) then Halt(1)
      else Halt(ErrorCode);
      end:
See also section 92.
This code is used in section 89.
```

28 MONITOR Mizar Parser §91

```
\langle Handle runtime error cases for monitor.pas 91 \rangle \equiv
  case RTErrorCode of
  800,804: DrawMessage('Library Corrupted', ');
  857: DrawMessage('Connection_Fault', '');
         { 900..999: DrawMessage('Mizar parameter overflow: '+IntToStr(RTErrorCode),"); }
  1255: DrawMessage('User_break', '');
  othercases if OverflowErroor then
       DrawMessage(`Mizar_parameter_overflow:_i` + IntToStr(RTErrorCode),``)
    else BugInProcessor
  endcases:
This code is used in section 90.
    The MizExitProc is a private "bail out" function.
\langle \text{Implementation for monitor.pas } 90 \rangle + \equiv
procedure MizExitProc:
  begin
  @\{@\&\$IFDEFIODEBUG@\}ExitProc \leftarrow \_ExitProc;
  Q{Q\&\$ELSEQ}_-Halt_-(ExitCode);
  @{@&$ENDIF@}
  end:
93. We use the MizExitProc to initialize the ExitProc pointer.
procedure InitExitProc:
  begin ExitProc \leftarrow @MizExitProc
  end;
94. Initializing Control. This is a heavily system dependent piece of code. There are two ways to
implement it (one way for Windows, another way for everyone else). Once we're done, we have to initialize
the _ExitProc and invoke InitCtrl.
  (Non-windows FreePascal implementation for InitCtrl 95)
  (Windows implemenation for InitCtrl 96)
  begin \_ExitProc \leftarrow ExitProc; InitCtrl;
  end.
95. \langle Non-windows FreePascal implementation for InitCtrl 95\rangle \equiv
  Q\{Q\&\$IFDEFFPCQ\} Q\{Q\&\$IFNDEFWIN32Q\}
  procedure CatchSignal(aSig:integer); cdecl;
    begin
      case aSiq of
           SIGINT, SIGQUIT, SIGTERM: begin CtrlCPressed \leftarrow true; RunTimeError(1255); end;
      endcases;
    end;
var NewSignal, OldSigInt: SignalHandler;
procedure InitCtrl;
  begin NewSignal \leftarrow SignalHandler(@CatchSignal); OldSigInt \leftarrow fpSignal(SIGINT, NewSignal);
  OldSigInt \leftarrow fpSignal(SIGQUIT, NewSignal); OldSigInt \leftarrow fpSignal(SIGTERM, NewSignal);
  end:
    0{0\&$ENDIF}0{0\&$ENDIF}0}
This code is used in section 94.
```

§96 Mizar Parser MONITOR 29

```
Microsoft breaks everything. This is a mess because of them.
\langle Windows implementation for InitCtrl 96\rangle \equiv
  @{@&$IFDEF WIN32@}
    (Windows implementation for CtrlSignal 99)
    @{@&$IFDEF FPC@}
       (FreePascal implementation of InitCtrl for Windows 97)
    @{@&$ENDIF @}
    @{@&$IFDEF DELPHI@}
       ⟨ Delphi implementation of InitCtrl for Windows 98⟩
    @{@&$ENDIF@}
  @{@&$ENDIF@}
This code is used in section 94.
     The FreePascal implementation is pretty succinct thanks to the libraries they provide.
\langle FreePascal implementation of InitCtrl for Windows 97\rangle \equiv
procedure InitCtrl;
  begin SetConsoleCtrlHandler(CtrlSignal, TRUE); end;
This code is used in section 96.
98. \langle \text{ Delphi implementation of } InitCtrl \text{ for Windows } 98 \rangle \equiv
procedure InitCtrl;
  var ConsoleMode, lConsoleMode: DWORD;
  \textbf{begin if } \textit{GetConsoleMode}(\textit{GetStdHandle}(\textit{STD\_INPUT\_HANDLE}), \textit{ConsoleMode}) \textbf{ then}
    begin lConsoleMode \leftarrow ConsoleMode \lor ENABLE\_PROCESSED\_INPUT;
         { Treat Ctrl+C as a signal }
    if SetConsoleMode(GetStdHandle(STD\_INPUT\_HANDLE), lConsoleMode) then
       begin SetConsoleCtrlHandler(@CtrlSignal, TRUE);
       end:
    end;
  end:
This code is used in section 96.
99. Windows requires a helper function CtrlSignal for this Microsoft mania.
\langle Windows implementation for CtrlSignal 99\rangle \equiv
  ⟨ FreePascal declaration of CtrlSignal for Windows 100⟩
  (Delphi declaration of CtrlSignal for Windows 101)
          {TRUE: do not call next handler in the queue, FALSE: call it }
  CtrlCPressed \leftarrow true; RunTimeError(1255); CtrlSignal \leftarrow true; \{ExitProcess(1); \}
This code is used in section 96.
       \langle FreePascal declaration of CtrlSignal for Windows 100\rangle \equiv
  @{@&$IFDEF FPC@}
function CtrlSignal(aSignal: DWORD): WINBOOL; stdcall;
    @{@&$ENDIF @}
This code is used in section 99.
101. \langle \text{ Delphi declaration of } Ctrl Signal \text{ for Windows } 101 \rangle \equiv
  @{@&$IFDEF DELPHI @}
function CtrlSignal(aSignal: DWORD): BOOL; cdecl;
  @{@&$ENDIF@}
This code is used in section 99.
```

30 ERROR HANDLING Mizar Parser $\S 102$

File 6

Error handling

```
102.
\langle \text{errhan.pas } 102 \rangle \equiv
  (GNU License 4)
unit errhan;
  interface
     ⟨Interface for errhan.pas 103⟩
  implementation
  uses mconsole, mizenv;
     ⟨Implementation for errhan.pas 106⟩
  end;
103. We have a few custom types and internal variables describing the state of the Mizar error handler.
\langle \text{Interface for errhan.pas } 103 \rangle \equiv
type Position = \langle Declare Position as record 104 \rangle;
  ErrorReport = \mathbf{procedure} \ (Pos: Position; ErrNr: integer);
const ZeroPos: Position = (Line: 0; Col: 0);
var CurPos: Position; { current position }
  ErrorNbr: integer; { current error number }
  PutError: ErrorReport = nil; { reporter for errors }
  RTErrorCode: integer = 0; { runtime error code }
  OverflowErroor: boolean = false; { overflow error? They're horrible, treat accordingly }
See also section 105.
This code is used in section 102.
104. Position is just a pair of integers recording the line and offset ("column").
\langle \text{ Declare } Position \text{ as } \mathbf{record} \quad 104 \rangle \equiv
  record Line, Col: integer
  end
This code is used in section 103.
```

 $\{105$ Mizar Parser ERROR HANDLING 31

105. And we just provide the public-facing functions and procedures. $\langle \text{Interface for errhan.pas } 103 \rangle + \equiv$ procedure Error(Pos : Position; ErrNr : integer); **procedure** *ErrImm*(*ErrNr* : *integer*); procedure WriteError(Pos : Position; ErrNr : integer); **procedure** OpenErrors(FileName : String); **procedure** AppendErrors(FileName : String); **procedure** EraseErrors: procedure CloseErrors; **procedure** OverflowError(ErrorCode: word); **procedure** Mizassert(ErrorCode: word; Cond: boolean); **procedure** RunTimeError(ErrorCode: word); The implementation begins as we would expect/hope. If we have a preferred error reporter already present in PutError, then we just use it. If we have toggled StopOnError to true, then we should end the program here (with a message). If we want to report an error at the CurrPos (current position), then we have a helper function do that for us. $\langle \text{Implementation for errhan.pas } 106 \rangle \equiv$ **procedure** Error(Pos: Position; ErrNr: integer); **begin** inc(ErrorNbr); if $@PutError \neq nil then PutError(Pos, ErrNr);$ if StopOnError then begin DrawMessage('Stopped_on_first_error', '); Halt(1); end; end; **procedure** ErrImm(ErrNr : integer): **begin** Error(CurPos, ErrNr); end: This code is used in section 102. We also can write errors to a file. This requires keeping track of the file (dubbed *Errors*) and whether it has been opened or not (in the Boolean condition *OpenedErrors*). Note this takes advantage of with to destructure Pos into a Line and Col for us. var Errors: text; { file name for errors file } OpenedErrors: boolean = false; { have we opened it yet? } **procedure** WriteError(Pos: Position; ErrNr: integer); **begin if** $\neg OpenedErrors$ **then** RunTimeError(2001); with Pos do WriteLn(Errors, Line, `` ', Col, `` ', ErrNr);

end;

32 ERROR HANDLING Mizar Parser $\S 108$

Opening an errors file. We can open an errors file, which will reset the ErrorNbr counter to zero and re-initialize CurPos to line 1 and column 1. When PutError is **nil**, we initialize it to be WriteError. **procedure** OpenErrors(FileName : String); **begin** if $ExtractFileExt(FileName) = ``then FileName <math>\leftarrow$ FileName + `.err'; $Assign(Errors, FileName); without_io_checking(Rewrite(Errors));$ if $IOResult \neq 0$ then begin DrawMessage('Can´´t⊔open⊔errors⊔file⊔´´´+ FileName + ´´´⊔for⊔writing´, ´´); halt(1); end; $OpenedErrors \leftarrow true : ErrorNbr \leftarrow 0$: with CurPos do **begin** $Line \leftarrow 1$; $Col \leftarrow 1$ end; if $@PutError = nil then PutError \leftarrow WriteError;$ end; 109. Appending errors to the errors file. **procedure** AppendErrors (FileName : String); **begin** $OpenedErrors \leftarrow true$; **if** ExtractFileExt(FileName) = ``**then** FileName ← FileName + `.err`; $Assign(Errors, FileName); ErrorNbr \leftarrow 0;$ with CurPos do **begin** $Line \leftarrow 1$; $Col \leftarrow 1$ end: without_io_checking(append(Errors)); if $ioresult \neq 0$ then Rewrite(Errors); end; We can also close the errors file and unset the Errors variable, "forgetting" where we logged the 110. errors. procedure EraseErrors; begin if OpenedErrors then **begin** $OpenedErrors \leftarrow false; close(Errors); erase(Errors);$ end; end; 111. We can also just close the errors file. procedure CloseErrors; begin if OpenedErrors then **begin** $OpenedErrors \leftarrow false; close(Errors);$ end;

112. Like I said, overflow errors are especially problematic. If/when they occur, we should just bail out immediately.

```
 \begin{array}{l} \textbf{procedure} \ \ \textit{OverflowError}(\textit{ErrorCode}: \textit{word}); \\ \textbf{begin} \ \ \textit{RTErrorCode} \leftarrow \textit{ErrorCode}; \ \ \textit{OverflowErroor} \leftarrow \textit{true}; \ \textit{RunError}(97); \\ \textbf{end}; \end{array}
```

end;

 $\S113$ Mizar Parser ERROR HANDLING 33

113. We have an assertion utility to check if a Cond is true. When it is false, we should report a runtime error.

```
 \begin{array}{ll} \textbf{procedure} \ \textit{MizAssert}(\textit{ErrorCode}:\textit{word}; \ \textit{Cond}:\textit{boolean}); \\ \textbf{begin} \ \textbf{if} \ \neg \textit{Cond} \ \textbf{then} \\ \textbf{begin} \ \textit{RTErrorCode} \leftarrow \textit{ErrorCode}; \ \textit{RunError}(98); \\ \textbf{end}; \\ \textbf{end}; \end{array}
```

114. Last, we have a catchall for runtime errors encountered.

```
\begin{array}{l} \textbf{procedure} \ \textit{RunTimeError}(\textit{ErrorCode}: word);\\ \textbf{begin} \ \textit{RTErrorCode} \leftarrow \textit{ErrorCode}; \ \textit{RunError}(99);\\ \textbf{end}; \end{array}
```

34 TIME UTILITIES Mizar Parser §115

File 7

Time utilities

```
This is another heavily "system dependent" library.
\langle \text{ mtime.pas } 115 \rangle \equiv
  (GNU License 4)
unit mtime;
  interface
     ⟨Interface for mtime.pas 116⟩
  implementation
     ⟨Implementation for mtime.pas 118⟩
  end;
116. \langle \text{Interface for mtime.pas } 116 \rangle \equiv
procedure TimeMark(\mathbf{var}\ W: longint);
function ElapsedTime(W : longint): longint;
procedure MUnpackTime(W : longint; var H, M, S, F : word);
function ReportTime(W : longint): String;
This code is used in section 115.
117. We also have one global variable tracking the start time.
var qStartTime: longint;
       The implementation begins with a rather thorny digression depending on which compiler we're using.
\langle \text{Implementation for mtime.pas } 118 \rangle \equiv
  ⟨Timing utilities uses for Delphi 119⟩
  ⟨Timing utilities uses for FreePascal 120⟩
See also section 121.
This code is used in section 115.
119. Delphi simply requires us to introduce a constant for milliseconds.
\langle\, {\rm Timing} \,\, {\rm utilities} \,\, {\bf uses} \,\, {\rm for} \,\, {\rm Delphi} \,\, {}_{119} \,\rangle \equiv
  @{@&$IFDEF DELPHI @}
  uses windows;
const cmSecs = 1000;
  @{@&$ENDIF@}
This code is used in section 118.
```

 $\{120 \quad \text{Mizar Parser} \quad \text{TIME UTILITIES} \quad 35$

```
FreePascal requires a bit more work, alas.
\langle Timing utilities uses for FreePascal 120\rangle \equiv
     @{@&$IFDEF FPC@}
     uses dos;
const cmSecs = 100;
type TSystemTime =
     record wHour: word;
     wMinute: word;
     wSecond: word;
     wMilliseconds: word:
     end:
procedure GetLocalTime (var aTime : TSystemTime);
     begin with aTime do GetTime(wHour, wMinute, wSecond, wMilliseconds);
     end:
     @{@&$ENDIF@}
This code is used in section 118.
121. Now we can happily plug along implementing the functions we need.
\langle \text{Implementation for mtime.pas } 118 \rangle + \equiv
function SystemTimeToMiliSec(const fTime: TSystemTime): longint;
     begin SystemTimeToMiliSec \leftarrow fTime.wHour * (3600 * cmSecs) + fTime.wMinute * longint (60 * cmSecs) + fTime.wminute * longin
               cmSecs) + fTime.wSecond * cmSecs + <math>fTime.wMilliseconds;
    end:
122. We "start the clock".
procedure TimeMark(\mathbf{var}\ W: longint);
     var SystemTime: TSystemTime;
     begin GetLocalTime(SystemTime); W \leftarrow SystemTimeToMiliSec(SystemTime);
     end;
              We measure the time lapse since we "started the clock".
function ElapsedTime(W : longint): longint;
     var T: longint; SystemTime: TSystemTime;
     begin GetLocalTime(SystemTime); T \leftarrow SystemTimeToMiliSec(SystemTime) - W;
     if T < 0 then T \leftarrow 86400 * cmSecs + T;
     ElapsedTime \leftarrow T;
     end;
124. We can transform an interval of time (in milliseconds) into hours, minutes, seconds, a fractional
amount of time.
procedure MUnpackTime(W : longint; var H, M, S, F : word);
     begin H \leftarrow W \text{ div } (3600 * cmSecs); M \leftarrow (W - H * 3600 * cmSecs) \text{ div } (60 * cmSecs);
     S \leftarrow (W - H * 3600 * cmSecs - M * 60 * cmSecs) div cmSecs;
     F \leftarrow W - H * 3600 * cmSecs - M * 60 * cmSecs - S * cmSecs;
     end;
```

36 TIME UTILITIES Mizar Parser $\S125$

125. When reporting time, we want to pad the time by a zero. This is standard conventional stuff (e.g., I have an appointment at 11:01 AM, not 11:1 AM).

```
 \begin{array}{l} \textbf{function } LeadingZero(w:word) \colon String; \\ \textbf{var } lStr \colon String; \\ \textbf{begin } Str(w:0,lStr); \\ \textbf{if } length(lStr) = 1 \textbf{ then } lStr \leftarrow \texttt{`O'} + lStr; \\ LeadingZero \leftarrow lStr; \\ \textbf{end}; \end{array}
```

126. Reporting time transforms a time interval (measured in milliseconds) into a human readable String.

```
function ReportTime(W:longint): String;
var\ H, M, S, F: word;\ lTimeStr:\ String;
begin\ MUnpackTime(ElapsedTime(W), H, M, S, F);
if\ F \geq (cmSecs\ div\ 2)\ then\ inc(S);
if\ H \neq 0\ then
begin\ Str(H, lTimeStr);\ lTimeStr \leftarrow lTimeStr + `.` + LeadingZero(M)
end
else\ Str(M:2, lTimeStr);
ReportTime\ \leftarrow lTimeStr + `:` + LeadingZero(S);
end;
```

127. When we run the program, we should mark the time.

```
\begin{array}{ll} \textbf{begin} & \textit{TimeMark}(\textit{gStartTime}); \\ \textbf{end.} \end{array}
```

File 8

Arbitrary precision arithmetic

128. Specifically, arbitrary precision arithmetic on *integers* and *rational complex* numbers. integers are represented as Strings of digits.

Note:

- (1) The naming convention dictates all functions suffixed with _XXX presuppose the arguments are positive.
- (2) Also there are no checks whether the parameters contain only digits (and an optional sign "-").
- (3) Further, DEBUGNUM is a conditional variable that can be used (with DEBUG) for testing.

```
\langle \text{ numbers.pas } 128 \rangle \equiv
  (GNU License 4)
unit numbers;
  interface
    (Basic arithmetic operations declarations 129)
     Types for arbitrary-precision arithmetic 130
     Zero and units for arbitrary-precision 131
     Rational arithmetic declarations 132
     (Predicate declarations for arbitrary-precision arithmetic 133)
     (Declare public complex-valued arbitrary precision arithmetic 134)
     (Declare public comparison operators for arbitrary-precision numbers 135)
  implementation
  uses mizenv
    @{@&$IFDEF CH_REPORT@}, req_info, prephan, builtin@{@&$ENDIF@}
      mdebug , info end_mdebug;
  (Trim leading zeros from arbitrary-precision integers 136)
   Check if arbitrary-precision integers are zero 137
   Absolute value for an arbitrary-precision number 138
   Test if one arbitrary-precision number is less than or equal to another 139)
  Arithmetic for arbitrary-precision integers 143
  (Arbitrary-precision rational arithmetic 160)
  (Complex-rational arbitrary-precision arithmetic 169)
```

function RationalGT (const r1, r2: Rational): boolean;

This code is used in section 128.

38

```
\langle Predicate declarations for arbitrary-precision arithmetic 133\rangle \equiv
function IsintegerNumber (const z: RComplex): boolean;
function IsNaturalNumber(const z: RComplex): boolean;
function IsPrimeNumber(const z: RComplex): boolean;
function AreEqComplex(const z1, z2: RComplex): boolean;
function IsEqWithInt(const z: RComplex;
                                 n: longint): boolean;
function IsRationalLE (const z1, z2: RComplex): boolean;
function IsRationalGT (const z1, z2: RComplex): boolean;
This code is used in section 128.
134. (Declare public complex-valued arbitrary precision arithmetic 134) \equiv
function IntToComplex(x:integer): RComplex;
function ComplexAdd(const z1, z2: RComplex): RComplex;
function ComplexSub(const z1, z2: RComplex): RComplex:
function ComplexNeg(const z: RComplex): RComplex;
function ComplexMult(const z1, z2: RComplex): RComplex;
function ComplexInv(const z: RComplex): RComplex;
function ComplexDiv(const z1, z2: RComplex): RComplex;
function ComplexNorm(const z: RComplex): Rational;
This code is used in section 128.
135. (Declare public comparison operators for arbitrary-precision numbers 135) \equiv
function CompareInt(X1, X2 : Longint): integer;
function CompareIntStr(X1, X2 : String): integer;
function CompareComplex(const z1, z2: RComplex): integer;
This code is used in section 128.
136. If we are given single character String consisting of zero or the empty String, then we are done.
  If we are given anything else, we find the first index (from the left) of a nonzero character. Then we create
a copy of the subString starting from the first nonzero digit to the rest of the String.
\langle Trim leading zeros from arbitrary-precision integers 136\rangle \equiv
function trimlz(a : String): String;
  var i: integer;
  begin if (a = `0`) \lor (a = ``) then trimlz \leftarrow a
  else begin i \leftarrow 0;
    repeat i \leftarrow i + 1;
      if a[i] \neq 0 then break;
    until i = length(a);
    trimlz \leftarrow copy(a, i, length(a));
    end;
  end:
This code is used in section 128.
```

Mizar Parser

40

end;

This code is used in section 128.

137. First, we check if a starts with "-0". If so, replace a with 0. Then we do the same thing with b. We invoke *trimlz* on a and store the result in a1. If $a1 \neq a$, then we update $a \leftarrow a1$. Then we do likewise on b. \langle Check if arbitrary-precision integers are zero $137 \rangle \equiv$ **procedure** $checkzero(\mathbf{var}\ a, b: String);$ **var** *a1*, *b1*: *String*; begin if copy(a, 1, 2) = `-0' then @{@&\$IFDEF DEBUGNUM @} WriteLn(infofile, `a=-0`); $0{0\&\$ENDIF0}a \leftarrow 0;$ end; **if** copy(b, 1, 2) = `-0` **then** begin @{@&\$IFDEF DEBUGNUM @} WriteLn(infofile, 'b=-0'); @{@&\$*ENDIF* @} $b \leftarrow 0$; end; $a1 \leftarrow trimlz(a)$; if $a1 \neq a$ then begin @{@&\$IFDEF DEBUGNUM @} WriteLn(infofile, 'ZEROS1: ', a); $0{0\&\$ENDIF 0}a \leftarrow a1;$ end; $b1 \leftarrow trimlz(b);$ if $b1 \neq b$ then begin @{@&\$IFDEF DEBUGNUM @} WriteLn(infofile, `ZEROS2: `, b); $0{0\&\$ENDIF 0}b \leftarrow b1;$ end: end; This code is used in section 128. 138. Since arbitrary precision numbers (as Strings) are negative if they begin with a leading "-" character, it is easy to obtain the absolute value (just delete the minus sign). \langle Absolute value for an arbitrary-precision number $138 \rangle \equiv$ function Abs(a:String): String; begin if length(a) > 0 then **if** a[1] = - **then** delete(a, 1, 1); $Abs \leftarrow a;$

139. When checking $a \leq b$ for two non-negative integers, written as Strings (without leading zeros) you can check if the length of a is less than the length of b.

If the length of b is less than the length of a, then b < a.

When the length of the two Strings are equal, use lexicographic ordering to determine which is less.

 \langle Test if one arbitrary-precision number is less than or equal to another 139 \rangle \equiv **function** $_leg(a, b : String)$: boolean;

```
var i, x, y, z: integer;
begin @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, '_leq(',a,',',b,')');
Q\{Q\&\$ENDIFQ\} checkzero (a,b);
if length(a) < length(b) then \_leq \leftarrow true
else if length(a) > length(b) then \_leq \leftarrow false
  else begin for i \leftarrow 1 to length(a) do
       begin val(a[i], x, z); val(b[i], y, z);
       if x > y then
          begin \_leq \leftarrow false; exit;
          end;
       if x < y then
          begin \_leq \leftarrow true; exit;
          end;
       end;
     \_leg \leftarrow true;
     end;
end:
```

140. Now the *general* case is when a and b are arbitrary-precision *integers*. If a starts with a minus sign and b starts with a minus sign, then test if $a \ge b$.

When a does not start with a minus sign, but b does start with a minus sign, then we're done: b < a. When neither a nor b starts with a minus sign, then we use $_{-leq}(a,b)$ to determine the result.

function leg(a, b : String): boolean;

This code is used in section 128.

```
begin 0{0\&\$IFDEFDEBUGNUM} 0} WriteLn(infofile, `leq(`,a, `, `,b, `)`); 0{0\&\$ENDIF} 0} checkzero(a,b); if a=b then leq \leftarrow true else begin if (a[1]=`-`) \wedge (b[1]\neq `-`) then leq \leftarrow true; if (a[1]=`-`) \wedge (b[1]=`-`) then leq \leftarrow \neg leq(abs(a),abs(b)); if (a[1]\neq `-`) \wedge (b[1]=`-`) then leq \leftarrow false; if (a[1]\neq `-`) \wedge (b[1]\neq `-`) then leq \leftarrow leq(a,b); end; end;
```

141. Testing if a > b is simply testing if b < a after normalizing the Strings.

```
 \begin{array}{l} \textbf{function} \ \ geq(a,b:String) \colon boolean; \\ \textbf{begin} \ \ @\{@\&\$IFDEFDEBUGNUM\,@\} \ WriteLn(infofile, `geq(`,a,`,`,b,`)`); \\ @\{@\&\$ENDIF\,@\} \ checkzero\,(a,b); \\ geq \leftarrow (\neg leq(a,b)) \lor (a=b); \\ \textbf{end}; \end{array}
```

Mizar Parser

42

end;

```
142. Similarly, we may check if a < b by testing a \neq b and a \leq b.
function le(a, b : String): boolean;
  begin @{@\&\$IFDEFDEBUGNUM@}WriteLn(infofile, `le(`,a,`,`,b,`)`);
  @{@\&\$ENDIF@} checkzero(a,b); le \leftarrow (a \neq b) \land (leq(a,b));
  end;
function gt(a, b : String): boolean;
  \textbf{begin @{@&$} \textit{IFDEF DEBUGNUM @}} \ \textit{WriteLn(infofile, `gt(`,a,`,`,b,`)`)}; \\
  \texttt{Q{Q&$ENDIFQ}} \ checkzero \ (a,b); \ \ gt \leftarrow \neg leq \ (a,b);
```

143. Arithmetic operations. Now we get to some interesting bits.

We have Add for the addition of two non-negative integers. The basic strategy is to go digit-by-digit, use the PASCAL-provided integer arithmetic, manually "carrying" 1 if necessary.

The basic strategy is to initialize a1 to be the larger of the two numbers, and b1 to the smaller of the two numbers. Then generically we will have

We will separate this out into two sums. First we compute

$$\frac{a_m \, a_{m-1} \dots a_1}{+ b_m \, b_{m-1} \dots b_1} \\
\underline{-c_{m+1} \, r_m \, r_{m-1} \dots r_1} \tag{143.2}$$

Then we will compute

$$\frac{a_n \dots a_{m+1}}{r_{m+1} r_n \dots r_{m+1}}$$
(143.3)

The result is assembled from the digits $r_{n+1}r_n\cdots r_1$.

 \langle Arithmetic for arbitrary-precision integers $\,$ 143 \rangle \equiv

```
function \_Add(a, b : String): String;
  var c, x, y, z, v: integer; i: integer; a1, b1, s, r: String;
  begin a1 \leftarrow a; b1 \leftarrow b;
  @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, `_Add(`, a1, `, `, b1, `)`);
  Q\{Q\&\$ENDIFQ\} checkzero (a1,b1);
  if length(a1) < length(b1) then
     begin s \leftarrow b1; b1 \leftarrow a1; a1 \leftarrow s; end;
  r \leftarrow \text{``}; \ c \leftarrow 0;
  begin for i \leftarrow 0 to length(b1) - 1 do { step 1, Eq (143.2) }
     begin val(a1 [length(a1) - i], x, z); val(b1 [length(b1) - i], y, z);
     if x + y + c > 9 then
        begin v \leftarrow (x+y+c) - 10; c \leftarrow 1;
     else begin v \leftarrow x + y + c; c \leftarrow 0;
        end;
     Str(v,s); r \leftarrow s + r;
     end:
  for i \leftarrow length(b1) to length(a1) - 1 do { step 2, Eq (143.3) }
     begin val(a1[length(a1) - i], x, z);
     if x + c > 9 then
        begin v \leftarrow (x+c) - 10; c \leftarrow 1;
     else begin v \leftarrow x + c; c \leftarrow 0;
        end;
     Str(v,s); r \leftarrow s + r;
     end;
  if c = 1 then r \leftarrow 1 + r;
  \_Add \leftarrow trimlz(r);
  end:
```

See also sections 146 and 151.

This code is used in section 128.

This code is used in section 144.

```
Subtraction is a bit trickier, because of the "borrowing" operation.
  Also note that \_Sub(a,b) will start by computing a_1 \leftarrow \max(a,b) and b_1 \leftarrow \min(a,b), then return a_1 - b_1.
function \_Sub(a, b : String): String;
  var x, y, z, v: integer; i: integer; a1, b1, s, r: String;
     \langle Borrow 1 \text{ for } Sub_- 145 \rangle
     begin a1 \leftarrow a; b1 \leftarrow b;
     @{@\&\$IFDEF\ DEBUGNUM\ @}\ WriteLn(infofile, `\_Sub(`, a1, `, `, b1, `)`);
     Q{Q\&\$ENDIFQ}checkzero(a1,b1);
     if \neg leg(b1, a1) then begin s \leftarrow b1; b1 \leftarrow a1; a1 \leftarrow s; end;
     r \leftarrow  ;
        begin
          for i \leftarrow 0 to length(b1) - 1 do
             begin val(a1 [length(a1) - i], x, z); val(b1 [length(b1) - i], y, z);
             if x < y then
                begin borrow(length(a1) - i); x \leftarrow x + 10; end;
             v \leftarrow x - y; Str(v, s); r \leftarrow s + r;
             end;
          for i \leftarrow length(a1) - length(b1) downto 1 do
             begin r \leftarrow a1[i] + r; end;
        end;
     \_Sub \leftarrow trimlz(r);
  end;
145.
       This is a private "helper function" for subtraction.
\langle \text{Borrow 1 for } Sub_- \text{ 145} \rangle \equiv
\mathbf{procedure}\ Borrow(k:integer);
  var xx, zz: integer; sx: String;
  begin val(a1[k-1], xx, zz);
  if xx \ge 1 then
     begin xx \leftarrow xx - 1; Str(xx, sx); a1[k-1] \leftarrow sx[1];
  else begin a1[k-1] \leftarrow \texttt{`9'}; borrow(k-1);
     end;
  end;
```

146. Multiplication. Multiplication of a by b works digit-by-digit, in the sense that for each digit b_i of b, we need to multiply a by b_i . The function $_Mul1$ does this.

```
\langle Arithmetic for arbitrary-precision integers 143 \rangle + \equiv
function _Mul1 (a : String; y : integer): String;
  var c, x, z, v: integer; i: integer; s, r: String;
     Q{Q\&\$IFDEFDEBUGNUM} Q{WriteLn(infofile, `\_Mul1(`,a,`,`,y,`)`)};
     0{0\&$ENDIF}0}r \leftarrow ; c \leftarrow 0;
     begin
        for i \leftarrow 0 to length(a) - 1 do
          begin val(a[length(a) - i], x, z);
          if x * y + c > 9 then
             begin v \leftarrow (x * y + c) \mod 10; c \leftarrow (x * y + c) \operatorname{div} 10; end
           else begin v \leftarrow x * y + c; c \leftarrow 0; end;
           Str(v,s); r \leftarrow s + r;
          end:
        if c \neq 0 then
          begin Str(c, s); r \leftarrow s + r; end;
     end;
     \_mul1 \leftarrow trimlz(r);
  end;
```

147. Then multiplication proper amounts to decomposing b into its decimal expansion $\sum_k b_k 10^k$ and computing $(a \times b_k)10^k$.

```
function \_Mul(a, b : String): String;
  var y, z: integer; i, j: integer; a1, b1, s, r: String;
  begin
     a1 \leftarrow a; b1 \leftarrow b;
     @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, `_Mul(`, a1, `, `, b1, `));
     Q{Q\&\$ENDIFQ}checkzero(a1,b1);
     if length(a1) < length(b1) then
        begin s \leftarrow b1; b1 \leftarrow a1; a1 \leftarrow s; end;
     r \leftarrow \texttt{`0'};
     for i \leftarrow 0 to length(b1) - 1 do
        begin val(b1[length(b1) - i], y, z); s \leftarrow \_mul1(a1, y);
        for j \leftarrow 0 to i - 1 do s \leftarrow s + \text{`0'};
        r \leftarrow \_Add(r,s);
        end;
     \_Mul \leftarrow trimlz(r);
  end;
```

Mizar Parser

if $_leq(_mul(b,r),a)$ then

function $_Div(a, b : String)$: String;

begin $_div1 \leftarrow trimlz(r); exit; end;$

148. Division. The basic design is similar to multiplication. We will try to divide a by $b \times 10^k$ (which is zero whenever $b \times 10^k > a$). **function** $_Div1(a, b : String)$: String; var i: integer; r: String; begin $Q{Q\&\$IFDEFDEBUGNUMQ}WriteLn(infofile, `_Div1(`,a,`,`,b,`)`);$ $Q{Q\&$ENDIFQ} checkzero(a, b);$ if $\neg leq(b, a)$ then $_div1 \leftarrow \texttt{`0'}$ else for $i \leftarrow 9$ downto 1 do **begin** Str(i,r);

149.

end;

end;

end;

46

```
var z, c, i: integer; s, r, rs: String; b\_GPC: boolean;
   (Get the next digit for dividing arbitrary-precision integers 150)
begin
   @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, `_Div(`, a, `, `, b, `) `);
   Q{Q\&$ENDIFQ} checkzero(a, b);
  if a = b then _{-}div \leftarrow 11
   else if \neg leq(b, a) then _{-}div \leftarrow ´0´
  else begin s \leftarrow \ \ ; \ r \leftarrow \ \ ; \ z \leftarrow 1;
     for i \leftarrow 1 to length(b) do s \leftarrow s + a[i];
     if \neg leq(b, s) then
        begin s \leftarrow s + a[length(b) + 1]; \ z \leftarrow length(b) + 1; end
     else begin z \leftarrow length(b); end;
     repeat rs \leftarrow \_div1(s,b); r \leftarrow r + rs; gets; b\_GPC \leftarrow \_leq(b,s);
     until \neg b_{-}GPC;
      _{\mathbf{div}} \leftarrow trimlz(r);
   end;
```

```
150. \langle Get the next digit for dividing arbitrary-precision integers 150 \rangle \equiv
procedure qets;
  var j: integer;
  begin c \leftarrow 1; s \leftarrow \_Sub(s,\_mul(rs,b));
  if (s = 0) \land (trimlz(copy(a, z + c, length(a))) = 0) then
     begin @\{@\&\$IFDEF\,DEBUGNUM\,@\}\,WriteLn\,(infofile, `Rewriting_zeros: `, <math>copy(a, z + c, length(a)));
     \mathbb{Q}_{\omega}ENDIF_{\omega}r \leftarrow r + copy(a, z + c, length(a)); exit;
     end:
  if z + 1 \leq length(a) then
     begin s \leftarrow s + a[z+1]; inc(c);
     if (\neg leq(b,s)) then r \leftarrow r + \text{`0'};
     end;
  while (\neg leq(b, s)) \land (z + c \leq length(a)) do
     begin s \leftarrow s + a[z + c]; inc(c);
     if (\neg leq(b, s)) then r \leftarrow r + \text{`0'};
     end;
  z \leftarrow z + c - 1;
  end; \{gets\}
This code is used in section 149.
151. Modulo. We can compute a \mod b by observing if a < b then we should obtain a. Otherwise, we
should compute r \leftarrow a \operatorname{\mathbf{div}} b, then a - rb is a \operatorname{\mathbf{mod}} b.
\langle Arithmetic for arbitrary-precision integers 143 \rangle + \equiv
function \_Mod(a, b : String): String;
  var r: String;
  begin Q(Q&FDEFDEBUGNUMQ)WriteLn(infofile, `\_Mod(`, a, `, `, b, `)`);
  Q{Q\&$ENDIFQ}checkzero(a,b);
  if le(a,b) then r \leftarrow a
  else r \leftarrow \_Sub(a, \_mul(b, \_div(a, b)));
  \_Mod \leftarrow trimlz(r);
  @\{0\&\$IFDEF\ DEBUGNUM\ @\}\ WriteLn(infofile, `End_\_Mod:`,r);
  @{@&$ENDIF@}
  end;
```

Mizar Parser

48

152. Greatest common divisor. We can compute gcd(a,b) first by setting $a_1 \leftarrow |a|$ and $b_1 \leftarrow |b|$ (since gcd(a, b) = gcd(|a|, |b|). Then we handle the special cases: (1) $a_1 = 1$ or $b_1 = 1$, then $gcd(a_1, b_1) = 1$ (2) $a_1 = 0$ and $b_1 \neq 0$, then $gcd(a_1, b_1) = b_1$ (3) $a_1 \neq 0$ and $b_1 = 0$, then $gcd(a_1, b_1) = a_1$ (4) $a_1 = b_1$, then $gcd(a_1, b_1) = a_1$ Otherwise, we end up in the default case, which is handled by the **while** loop. function GCD(a, b : String): String; label ex: var a1, b1, p, r: String; **begin** $a1 \leftarrow a$; $b1 \leftarrow b$; $Q{Q\&\$IFDEF\ DEBUGNUM\ Q}\ WriteLn(infofile, `GCD(`, a1, `, `, b1, `)`);$ $\mathbb{Q}\{\mathbb{Q}_{s}=NDIF\mathbb{Q}\} checkzero(a1,b1); a1 \leftarrow abs(a1); b1 \leftarrow abs(b1);$ **if** $(a1 = 1) \lor (b1 = 1)$ **then** begin $r \leftarrow 1$; goto ex; end; if $(a1 = `0`) \land (b1 \neq `0`)$ then begin $r \leftarrow b1$; goto ex; end; if $(b1 = `0`) \land (a1 \neq `0`)$ then begin $r \leftarrow a1$; goto ex; end; if a1 = b1 then begin $GCD \leftarrow a1$; $r \leftarrow a1$; goto ex; end; while gt(b1, 0) do **begin** $p \leftarrow b1$; $b1 \leftarrow \mod(a1, b1)$; $a1 \leftarrow p$ **end**; $r \leftarrow a1$; $ex: GCD \leftarrow r;$ $@\{@\&\$IFDEF\ DEBUGNUM\ @\}\ WriteLn(infofile, `End_{\sqcup}GCD: `, r);$ @{@&\$*ENDIF*@} end; **Least common multiple.** We recall lcm(a, b) = |ab|/gcd(|a|, |b|). function LCM(a, b : String): String; **var** *a1*, *b1*, *r*: *String*; **begin** $a1 \leftarrow a$; $b1 \leftarrow b$; $Q{Q\&\$IFDEFDEBUGNUMQ}WriteLn(infofile, `LCM(`, a1, `, `, b1, `)`);$ $\mathbb{Q}\{\mathbb{Q}_{s}=NDIF\mathbb{Q}\}\ checkzero(a1,b1);\ a1 \leftarrow abs(a1);\ b1 \leftarrow abs(b1);$ $r \leftarrow Diva(Mul(a1,b1), GCD(a1,b1)); LCM \leftarrow r;$

 $@\{@\&\$IFDEFDEBUGNUM\,@\}WriteLn(infofile, `End_LCM:`,r);$

@{@&\$*ENDIF*@}

end;

end:

```
154. Addition. This is a bit obfuscated with the reliance of goto ex, but the basic idea is (recalling that
\_Sub(a,b) calculates \max(a,b) - \min(a,b) for a \ge 0 and b \ge 0:
(1) If a < 0 and b < 0, then a + b = -(|a| + |b|)
(2) Else if a \ge 0 and b \ge 0, then a + b is computed using Add
(3) Else if a < 0 and b \ge 0, then we have two cases
   (i) If |a| \ge b, compute a + b = -(|a| - b)
   (ii) Otherwise, a + b = b - |a|
(4) Else if a \ge 0 and b < 0, then a + b = a - |b|
(5) Otherwise, when a \ge 0 and b \ge 0, a + b is computed using Add.
function Add(a, b : String): String;
  label ex;
  var r: String;
  begin @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, `Add(`,a,`,`,b,`)`);
  Q{Q\&$ENDIF Q} checkzero(a, b);
  if (a[1] = `-`) \wedge (b[1] = `-`) then
     begin r \leftarrow \text{`-'} + Add(abs(a), abs(b));
    if r = \text{`-0'} then r \leftarrow \text{`0'};
     goto ex;
     end;
  if (a[1] \neq \texttt{`-'}) \land (b[1] \neq \texttt{`-'}) then
     begin r \leftarrow Add(a, b); goto ex; end;
  if (a[1] = `-`) \wedge (b[1] \neq `-`) then
    if qt(abs(a),b) then
       begin r \leftarrow `-` + \_Sub(abs(a), b);
       if r = \text{`-0'} then r \leftarrow \text{`0'};
       goto ex;
       end
     else begin r \leftarrow \_Sub(abs(a), b); goto ex; end;
  if (a[1] \neq `-`) \land (b[1] = `-`) then
    if gt(abs(b), a) then
       begin r \leftarrow `-` + \_Sub(abs(b), a);
       if r = -0 then r \leftarrow 0;
       goto ex;
       end
     else begin r \leftarrow \_Sub(abs(b), a); goto ex; end;
ex: Add \leftarrow r;
  @\{@\&\$IFDEF\ DEBUGNUM\ @\}\ WriteLn(infofile, `End\_Add: `,r);
  @{@&$ENDIF@}
```

50

155. Subtraction. Now, given two arbitrary precision integers, we can compute their difference. Again, **goto** *ex* obfuscates the flow here, but the basic logic is:

```
(1) If a < 0 and b > 0, then a - b = -(|a| + b)
(2) Else if a \ge 0 and b < 0, then a - b = a + |b|
(3) Else if a < 0 and b < 0, then we have two cases
   (i) If |a| > |b|, then a - b = -(|a| - |b|)
   (ii) Otherwise |a| \le |b|, so a - b = |a| - |b|
(4) Else if a \ge 0 and b \ge 0, then we have two cases
   (i) If b > a, then a - b = -(b - a)
   (ii) Otherwise compute a - b using \_Sub(a, b)
Testing if x < 0 is done by checking \operatorname{sgn}(x) = -1, and x \ge 0 tests if \operatorname{sgn}(x) \ne -1.
function Sub(a, b : String): String;
  label ex;
  var r: String;
  begin @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, `Sub(`,a,`,`,b,`)`);
  Q{Q\&$ENDIFQ}checkzero(a,b);
  if (a[1] = \text{`-'}) \land (b[1] \neq \text{`-'}) then
     begin r \leftarrow `-` + \_Add(abs(a), b);
     if r = -0 then r \leftarrow 0;
     goto ex;
     end:
  if (a[1] \neq `-`) \wedge (b[1] = `-`) then
     begin r \leftarrow Add(a, abs(b)); goto ex; end;
  if (a[1] = `-`) \wedge (b[1] = `-`) then
     if gt(abs(a), abs(b)) then
       begin r \leftarrow \text{`-'} + \text{\_}Sub(abs(a), abs(b));
       if r = -0 then r \leftarrow 0;
       goto ex;
       end
     else begin r \leftarrow \_Sub(abs(a), abs(b)); goto ex; end;
  if (a[1] \neq \texttt{`-'}) \land (b[1] \neq \texttt{`-'}) then
     if qt(b,a) then
       begin r \leftarrow `-` + \_Sub(b, a);
       if r = -0 then r \leftarrow -0;
       goto ex;
       end
     else begin r \leftarrow \_Sub(a, b); goto ex; end;
ex: Sub \leftarrow r:
  @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, 'End_Sub: ', r);
  @{@&$ENDIF@}
  end;
```

156. Multiplication of arbitrary-precision integers. We calculate the product of a with b by handling the case where $sgn(a) \neq sgn(b)$ as $ab = -|a| \cdot |b|$. Otherwise we can just rely on the $_{-}Mul(a,b)$ to do our work.

```
function Mul(a,b:String): String;
label ex;
var r: String;
begin @\{@\&\$IFDEFDEBUGNUM@\}WriteLn(infofile, `Mul(`,a,`,`,b,`)`);
@\{@\&\$ENDIF@\}checkzero(a,b);
if ((a[1] = `-`) \wedge (b[1] \neq `-`)) \vee ((a[1] \neq `-`) \wedge (b[1] = `-`)) then
begin r \leftarrow `-` + \_Mul(abs(a), abs(b));
if r = `-0` then r \leftarrow `0`;
end
else r \leftarrow \_Mul(abs(a), abs(b));
ex: Mul \leftarrow r;
@\{@\&\$IFDEFDEBUGNUM@\}WriteLn(infofile, `End\_Mul:`,r);
@\{@\&\$ENDIF@\}
end;
```

157. DivA. This is the division for arbitrary-precision integers. Like multiplication, we handle the case $sgn(a) \neq sgn(b)$ by computing a/b = -|a|/|b|.

```
function DivA(a,b:String): String;
label ex;
var r: String;
begin Q\{Q&\$IFDEFDEBUGNUMQ\}WriteLn(infofile, `DivA(`,a,`,`,b,`)`);
Q\{Q\&\$ENDIFQ\}checkzero(a,b);
if ((a[1] = `-`) \land (b[1] \neq `-`)) \lor ((a[1] \neq `-`) \land (b[1] = `-`)) then
begin r \leftarrow `-` + Div(abs(a), abs(b));
if r = `-0` then r \leftarrow `0`;
end
else r \leftarrow Div(abs(a), abs(b));
ex: DivA \leftarrow r;
Q\{Q\&\$IFDEFDEBUGNUMQ\}WriteLn(infofile, `End_DivA:`,r);
Q\{Q\&\$ENDIFQ\}
end;
```

Mizar Parser

52

Testing for primality. We can test if a given arbitrary-precision integer is prime or not. Specifically, we restrict attention to *positive* integers.

The while loop calculates Mul(i,i) because Fermat observed we only need to check numbers up to $\lceil \sqrt{x} \rceil$ as prime factors of x. But this calulation is a bit costly. This could be approximated by taking the length of the underlying String n = |s| and looking at the leading $\lceil n/2 \rceil$ digits s_{lead} . It's not hard to see that the number x_{lead} described by s_{lead} satisfies $x_{\text{lead}}^2 \geq x$.

```
{\bf function}\ {\it IsPrime} (a:String) \hbox{:}\ boolean;
  var i: String; r: boolean;
  begin if leq(`2`,a) then
     begin r \leftarrow true; i \leftarrow 2;
     while leq(Mul(i,i),a) do
       begin if GCD(a, i) = i then
          begin r \leftarrow false; break; end;
       i \leftarrow Add(i, 1);
       end;
     end
  else r \leftarrow false;
  IsPrime \leftarrow r;
  end;
        Divides relation. We can check if "x divides y" by testing if gcd(x,y) = |x|.
function Divides(a, b : String): boolean;
  var r: boolean;
  begin r \leftarrow GCD(a,b) = abs(a); Divides \leftarrow r;
  end:
```

160. Rational arithmetic. Now we begin the rational arithmetic "in earnest". The first thing to do is provide a way to compute the reduced form for a fraction, i.e.,

$$\frac{n}{d} = \frac{n/\gcd(n,d)}{d/\gcd(n,d)}$$

```
\langle Arbitrary-precision rational arithmetic 160\rangle \equiv
procedure RationalReduce(\mathbf{var}\ r: Rational);
  var lGcd: String;
  begin lGcd \leftarrow qcd(r.Num, r.Den); r.Num \leftarrow diva(r.Num, lGcd); r.Den \leftarrow diva(r.Den, lGcd);
  end:
```

161. Rational addition. We recall

This code is used in section 128.

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

```
function RationalAdd(const r1, r2: Rational): Rational;
  var lRes: Rational;
  begin lRes.Num \leftarrow Add(Mul(r1.Num, r2.Den), Mul(r1.Den, r2.Num));
  lRes.Den \leftarrow Mul(r1.Den, r2.Den); RationalReduce(lRes); RationalAdd \leftarrow lRes;
  end;
```

162. Rational subtraction. Similar to addition, but the numerator is ad - bc.

```
function RationalSub(\mathbf{const}\ r1, r2:\ Rational): Rational; var lRes:\ Rational; begin lRes.Num \leftarrow Sub(Mul(r1.Num, r2.Den), Mul(r1.Den, r2.Num)); lRes.Den \leftarrow Mul(r1.Den, r2.Den);\ RationalReduce(lRes);\ RationalSub \leftarrow lRes; end;
```

163. Negating a rational number amounts to multiplying the numerator by -1.

```
function RationalNeg(\mathbf{const}\ r1:\ Rational): Rational; var lRes:\ Rational; begin lRes.Num \leftarrow Mul(`-1`, r1.Num);\ lRes.Den \leftarrow r1.Den;\ RationalNeg \leftarrow lRes; end:
```

164. Multiplying rational numbers. This uses the school-book formula

$$\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$$

```
function RationalMult (const r1, r2: Rational): Rational; var\ lRes: Rational; begin\ lRes.Num \leftarrow Mul(r1.Num, r2.Num);\ lRes.Den \leftarrow Mul(r1.Den, r2.Den);\ RationalReduce(lRes);\ RationalMult \leftarrow lRes;\ end;
```

165. Inverting a rational number. This is easy, provided the numerator is nonzero. The convention is to make the numerator carry the sign of the number (so n/d has $n \in \mathbf{Z}$ while $d \in \mathbf{N}$).

When the rational number is zero, we simply take $0^{-1} = 0$ (as is conventional among proof assistants).

```
function RationalInv(const r: Rational): Rational;
```

```
var lRes: Rational;

begin if r.Num \neq `0` then

begin if le(r.Num, `0`) then lRes.Num \leftarrow Mul(`-1`, r.Den)

else lRes.Num \leftarrow r.Den;

lRes.Den \leftarrow Abs(r.Num);

end

else lRes \leftarrow RZero;

RationalInv \leftarrow lRes;

end;
```

166. Dividing rational numbers. We see that $r_1/r_2 = r_1 \times (r_2^{-1})$. That's the trick.

```
function RationalDiv(\mathbf{const}\ r1, r2:\ Rational):\ Rational;
begin RationalDiv \leftarrow RationalMult(r1, RationalInv(r2));
end;
```

167. Equality of rational numbers. Two rational numbers n_1/d_1 and n_2/d_2 are equal if $n_1 = n_2$ and $d_1 = d_2$. This assumes that both rational numbers are in reduced form.

```
function RationalEq(\mathbf{const}\ r1, r2:\ Rational):\ boolean;

begin RationalEq \leftarrow (r1.Num = r2.Num) \wedge (r1.Den = r2.Den);

end;
```

54

168. Testing inequality of rational numbers. We have $n_1/d_1 < n_2/d_2$ if $n_1d_2 < n_2d_1$.

Similarly, we have $n_1/d_1 \ge n_2/d_1$ is just the negation of $n_1/d_1 < n_2/d_2$. **But:** this is misleadingly called RationalGT instead of RationalGEQ.

```
function RationalLE (const r1, r2: Rational): boolean; begin RationalLE \leftarrow leq(Mul(r1.Num, r2.Den), Mul(r1.Den, r2.Num)); end; function RationalGT (const r1, r2: Rational): boolean; begin RationalGT \leftarrow \neg RationalLE(r1, r2); end;
```

169. Rational complex arbitrary-precision arithmetic. We now begin with $\mathbf{Q} + i\mathbf{Q} \subseteq \mathbf{C}$, the subset of complex-numbers where the real and imaginary parts are rational numbers.

We want to know when these numbers describe integers (i.e., the imaginary part is zero and the denominator of the real part is 1) and natural numbers (i.e., when furthermore the numerator of the real part is non-negative).

```
 \begin \begin
```

This code is used in section 128.

170. Equality of complex numbers. This amounts to checking if the real and imaginary parts are equal to each other as rational numbers.

```
function AreEqComplex (const z1, z2: RComplex): boolean;

begin\ AreEqComplex \leftarrow RationalEq(z1.Re, z2.Re) \land RationalEq(z1.Im, z2.Im); end;

function IsEqWithInt (const z: RComplex;

n: longint): boolean;

var\ s: String;

begin\ Str(n,s); IsEqWithInt \leftarrow (z.Im.Num = `0`) \land (z.Re.Num = s) \land (z.Re.Den = `1`); end;
```

171. "Inequalities". We "induce" the binary relations < and \ge on the subset $\{q + i0 \mid q \in \mathbf{Q}\} \subseteq \mathbf{C}$. Again, what we said earlier about RationalGT being badly named holds for IsRationalGT being badly named as well.

```
function IsRationalLE (const z1, z2: RComplex): boolean; begin \ IsRationalLE \leftarrow (z1.Im.Num = `0`) \land (z2.Im.Num = `0`) \land RationalLE(z1.Re, z2.Re); end; function IsRationalGT (const z1, z2: RComplex): boolean; begin \ IsRationalGT \leftarrow (z1.Im.Num = `0`) \land (z2.Im.Num = `0`) \land RationalGT(z1.Re, z2.Re); end;
```

172. Converting integers to complex numbers. We have a function to convert an integer $x \in \mathbf{Z}$ to be the complex number $(x/1) + \mathrm{i}(0/1) \in \mathbf{C}$.

```
function IntToComplex(x:integer): RComplex;
var lRes: RComplex;
begin lRes \leftarrow COne; \ lRes.Re.Num \leftarrow IntToStr(x); \ IntToComplex \leftarrow lRes;
end;
```

```
173. Adding complex numbers. We compute the sum of (x_1+iy_1) and x_2+iy_2 to be (x_1+x_2)+i(y_1+y_2). function ComplexAdd (const z1, z2: RComplex): RComplex; var lRes: RComplex; begin lRes. Re \leftarrow RationalAdd (z1. Re, z2. Re); lRes. lm \leftarrow RationalAdd (z1. lm, z2. lm); lRes lRe
```

174. Subtracting complex numbers. We find the difference of complex numbers componentwise.

```
function ComplexSub (const z1, z2: RComplex): RComplex; var lRes: RComplex; begin lRes. Re \leftarrow RationalSub (z1. Re, z2. Re); lRes. Im \leftarrow RationalSub (z1. Im, z2. Im); lRes (%$IFDEF CH_REPORT lRes) lRes (lRes); lRes); lRes (lRes); lRes); lRes);
```

175. Negating complex numbers. We negate a complex number -z by negating its real and imaginary parts.

```
 \begin{array}{l} \textbf{function} \ \ ComplexNeg(\textbf{const} \ z: \ RComplex): \ RComplex; \\ \textbf{var} \ \ lRes: \ RComplex; \\ \textbf{begin} \ \ lRes.Re \leftarrow RationalNeg(z.Re); \ \ lRes.Im \leftarrow RationalNeg(z.Im); \\ \textbf{@{@x$IFDEF} \ CH\_REPORT @} \ \ CHReport.Out\_NumReq2(rqRealNeg,z,lRes); \\ \textbf{@{@x$ENDIF @} \ } \ \ ComplexNeg \leftarrow lRes; \\ \textbf{end}; \\ \end{array}
```

176. Multiplying complex numbers. We use the usual formula

```
(x_1 + iy_1)(x_2 + iy_2) = (x_1x_2 - y_1y_2) + i(x_1y_2 + y_1x_2).
```

```
function ComplexMult (const z1, z2: RComplex): RComplex; var lRes: RComplex; begin if IsEqWithInt(z1,-1) then ComplexMult \leftarrow ComplexNeg(z2) else if IsEqWithInt(z2,-1) then ComplexMult \leftarrow ComplexNeg(z1) else begin lRes.Re \leftarrow RationalSub(RationalMult(z1.Re, z2.Re), RationalMult(z1.Im, z2.Im)); lRes.Im \leftarrow RationalAdd(RationalMult(z1.Re, z2.Im), RationalMult(z1.Im, z2.Re)); ComplexMult \leftarrow lRes; @{@&$IFDEF\ CH\_REPORT\ @}\ CHReport.Out\_NumReq3\ (rqRealMult, z1, z2, lRes); @{@&$ENDIF\ @} end; end;
```

56

177. Dividing complex numbers. We recall

$$\frac{x_1 + iy_1}{x_2 + iy_2} = \frac{(x_1 + iy_1)(x_2 - iy_2)}{x_2^2 + y_2^2}$$

This is the case for nonzero $z_2 \neq 0$. When we try to divide $z_1/0$, we return 0.

```
function ComplexDiv(const z1, z2: RComplex): RComplex;

var lDenom: Rational; lRes: RComplex;

begin lRes \leftarrow CZero;

with z2 do lDenom \leftarrow RationalAdd(RationalMult(Re, Re), RationalMult(Im, Im));

if lDenom.Num \neq `0` then

begin

lRes.Re \leftarrow RationalDiv(RationalAdd(RationalMult(z1.Re, z2.Re), RationalMult(z1.Im, z2.Im)),

lDenom);

lRes.Im \leftarrow RationalDiv(RationalSub(RationalMult(z1.Im, z2.Re), RationalMult(z1.Re, z2.Im)),

lDenom);

lDenom);
```

178. Inverting complex numbers. We can now calculate z^{-1} as just 1/z.

```
function ComplexInv(\mathbf{const}\ z:\ RComplex):\ RComplex;

begin ComplexInv \leftarrow ComplexDiv(COne, z); end;
```

179. Norm of complex numbers. The "norm" or *modulus* for a complex number is just the sum of the square of its components (well, the squareroot of this sum).

```
function ComplexNorm(\mathbf{const}\ z:\ RComplex):\ Rational;

begin ComplexNorm \leftarrow RationalAdd(RationalMult(Z.Re, Z.Re), RationalMult(Z.Im, Z.Im)); end;
```

180. Comparison functions. The remainder of numbers. pas defines functions which compares numbers. These must return a value in the set $\{-1,0,+1\}$ as a PASCAL *integer*.

```
function CompareInt(X1, X2 : Longint): integer; begin if X1 = X2 then CompareInt \leftarrow 0 else if X1 > X2 then CompareInt \leftarrow 1 else CompareInt \leftarrow -1; end; function CompareIntStr(X1, X2 : String): integer; begin if X1 = X2 then CompareIntStr \leftarrow 0 else if gt(X1, X2) then CompareIntStr \leftarrow 1 else CompareIntStr \leftarrow -1; end;
```

181. There is also a function to "compare" complex numbers. This treats a complex number

$$z = \frac{n_1}{d_1} + i\frac{n_2}{d_2}$$

as a tuple (n_1, d_1, n_2, d_2) then uses lexicographic ordering based on the components.

function CompareComplex(**const** z1, z2: RComplex): integer;

var *lInt*: *integer*;

begin $lInt \leftarrow CompareIntStr(z1.Re.Num, z2.Re.Num);$

if $lInt \neq 0$ then

begin $CompareComplex \leftarrow lInt; exit end;$

 $lInt \leftarrow CompareIntStr(z1.Re.Den, z2.Re.Den);$

if $lInt \neq 0$ then

begin $CompareComplex \leftarrow lInt; exit end;$

 $lInt \leftarrow CompareIntStr(z1.Im.Num, z2.Im.Num);$

if $lInt \neq 0$ then

begin $CompareComplex \leftarrow lInt$; exit **end**;

 $CompareComplex \leftarrow CompareIntStr(z1.Im.Den, z2.Im.Den);$

end;

58

Mizar Objects and Data Structures

This is one of the largest files in Mizar (it clocks in at 6594 lines of code). Its interface consists of 552 lines alone (roughly 1/13 of the file).

```
\langle \text{ mobjects.pas } 182 \rangle \equiv
  ⟨GNU License 4⟩
unit mobjects;
  interface
  uses numbers:
    (Public interface for mobjects.pas 184)
  implementation
  mdebug uses info;
  end_mdebug
  (Implementation for mobjects.pas 183)
  end;
```

We have an error method for situations when a method is not implemented, for example when there is no ordering operator when the user invokes MSortedCollection. Compare (§288).

```
\langle \text{Implementation for mobjects.pas } 183 \rangle \equiv
procedure Abstract1;
  begin RunError(211);
  end;
   \langle MObject \text{ implementation } 187 \rangle
   \langle MStrObj \text{ implementation } 192 \rangle
   \langle MList \text{ implementation } 196 \rangle
    MCollection implementation 220 \rangle
    MExtList implementation 236 \rangle
    MSortedList implementation 251 \rangle
    MSortedExtList implementation 268\rangle
    MSortedStrList implementation 282 \rangle
    MSortedCollection implementation 287
    String collection implementation 296
    MIntCollection implementation 300 \rangle
    Stacked object implementation 308
   String list implementation 310
   (Int relation implementation 351)
   ⟨ Partial integer function implementation 360⟩
    NatFunc implementation 380 \rangle
   (NatSeg implementation 398)
   \langle IntSequence implementation 403 \rangle
   \langle IntSet \text{ Implementation 419} \rangle
   Partial Binary integer Functions 430
  (Partial integers to Pair of integers Functions 448)
```

This code is used in section 182.

184. Constant parameters.

```
\langle \text{Public interface for mobjects.pas } 184 \rangle \equiv
const
         { Maximum MCollection size }
  MaxSize = 2000000;
  MaxCollectionSize = MaxSize  div SizeOf(Pointer);
     { Maximum MStringList size }
  MaxListSize = MaxSize \ \mathbf{div} \ (SizeOf(Pointer) * 2);
     { Maximum IntegerList size }
  MaxIntegerListSize = MaxSize \ div \ (SizeOf (integer));
     { MCollection error codes }
  coIndexError = -1; { Index out of range }
  coOverflow = -2; \{ Overflow \}
  coConsistentError = -3;
  coDuplicate = -5; \{ Duplicate \}
  coSortedListError = -6;
  coIndexExtError = -7;
See also sections 185, 186, 191, 194, 195, 219, 235, 250, 267, 281, 286, 295, 299, 307, 309, 340, 341, 350, 359, 379, 397, 402, 418,
     429, 447, and 461.
This code is used in section 182.
185. Type aliases.
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
type { String pointers }
  PString = \uparrow ShortString;
     { Character set type }
  PCharSet = \uparrow TCharSet;
  TCharSet = \mathbf{set} \ \mathbf{of} \ char;
     { General arrays }
  PByteArray = \uparrow TByteArray;
  TByteArray = array [0.32767] \text{ of } byte; \{32767 = 2^{15} - 1\}
  PWordArray = \uparrow TWordArray;
```

 $TWordArray = array [0..16383] of word; {16383 = 2^{14} - 1}$

60 BASE OBJECT Mizar Parser $\S186$

Section 9.1. BASE OBJECT

186. Object-oriented PASCAL is a bit crufty (like all Object-oriented ALGOL-descended languages). The base *MObject* "class" has a constructor, destructor, a clone function named *CopyObject*, and a "move" function called *MCopy*.

```
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
    { MObject base object }
  PObject = \uparrow MObject;
  ObjectPtr = PObject;
  MObject = \mathbf{object}
    constructor Init;
    procedure Free:
    destructor Done; virtual;
    function CopyObject: PObject;
    function MCopy: PObject; virtual;
  end;
187.
       Note that the VER70 conditional compilation only plays a role here, in MObject.Init.
  The constructor will initialize the memory allocated for the MObject to be zero.
\langle MObject \text{ implementation } 187 \rangle \equiv
    { MObject }
constructor MObject.Init;
    @{@&$IFDEF VER70 @}
    type Image = \mathbf{record} \ Link: \ word;
         Data: record
           end;
         end:
    @{@&$ENDIF@}
       begin
       @\{@\&\$IFDEF\ VER70\ @\}FillChar(Image(Self).Data, SizeOf(Self) - SizeOf(MObject), 0);
       @{@&$ENDIF@}
       end;
This code is used in section 183.
188. Destructor. The MObject.Free procdure frees all the memory allocated to the caller.
  The destructor is, well, what C++ programmers would call an "abstract method".
procedure MObject.Free;
  begin Dispose(PObject(@Self), Done); end;
destructor MObject.Done;
```

189. Copying an object allocates new memory using the Free PASCAL *GetMem* function, then *moves* the contents of the caller to the new region. It **does not** "copy" the contents of the caller to the new region. If we wanted to copy the contents of the caller, we should call something like *Fillchar* or *Fillword*.

It then returns a pointer to the newly allocated object.

begin end;

```
function MObject.CopyObject: PObject;

var lObject: PObject;

begin GetMem(lObject, SizeOf(Self)); Move(Self, lObject \uparrow, SizeOf(Self)); CopyObject \leftarrow lObject;

end;
```

 $\S190$ Mizar Parser BASE OBJECT 61

190. The virtual method for copying Mizar objects can be overridden by subclasses. But the default method is just CopyObject.

 $\begin{array}{ll} \textbf{function} \ \ \textit{MObject.MCopy: PObject;} \\ \textbf{begin} \ \ \textit{MCopy} \leftarrow \textit{CopyObject; end;} \end{array}$

62 MIZAR STRING OBJECT Mizar Parser §191

Section 9.2. MIZAR STRING OBJECT

```
191. Strings in Mizar amount to a wrapper around the underlying string data type.
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
     { Specyfic objects based on MObjects for collections }
  PStr = \uparrow MStrObj;
  MStrPtr = PStr;
  MStrObj = \mathbf{object} \ (MObject)
    fStr: String;
     constructor Init(const aStr: String);
  end;
192. Constructor. The constructor for a string object expects a string, and simply initializes its contents
to the given string.
\langle MStrObj \text{ implementation } 192 \rangle \equiv
     { Specyfic objects based on MObjects for collections }
constructor MStrObj.Init(const aStr: String);
  begin fStr \leftarrow aStr; end;
This code is used in section 183.
```

 $\{193 \quad \text{Mizar Parser} \quad \text{MIZAR LIST} \quad 63$

Section 9.3. MIZAR LIST

193. A *MList* is a dynamic array data structure, which represents a list using an array. We reserve an array whose length is referred to as its "Capacity" in the literature.

Not all of the underlying array is used by the user. The number of entries which are used by the dynamic array contents is referred to as its "Logical Size" (or just its *Size*) in the literature.

When the dynamic array is filled, it "grows"; i.e., it allocates a new array that's larger, and copies over the contents of its old array, then frees the old array. The growth factor is controlled by the GrowLimit(oldSize) value.

194. Review of pointers in Pascal. We have a few parameters needed for collections. Remember, if T is a type, then $\uparrow T$ is the type of pointers to T objects. If we want to have a pointer without referring to the type of the object referenced, we can use Pointer.

The @ operator is the "address of" operator. When setting a pointer p to point to something Foo, we have $p \leftarrow @Foo$.

The \uparrow operator is the "dereferencing" operator which is appended to a pointer identifier. When we want to update the object referenced by a pointer p, we have $p\uparrow \leftarrow newValue$.

```
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv \{ \text{MCollection types} \} 
PItemList = \uparrow MItemList;
MItemList = \mathbf{array} \ [0 ... MaxCollectionSize - 1] \ \mathbf{of} \ Pointer;
```

64 MIZAR LIST Mizar Parser $\S195$

```
195. A MList object is known as a dynamic array. Java programmers would know that as an ArrayList.
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
    { MList object }
  PList = \uparrow MList;
  MListPtr = PList;
  MList = \mathbf{object} (MObject)
    Items: PItemList; { Contents of dynamic array }
    Count: integer; { Logical size of dynamic array }
    Limit: integer; { Capacity of dynamic array }
    constructor Init(ALimit : integer);
    constructor MoveList(var aAnother : MList);
    constructor CopyList(var aAnother : MList);
    destructor Done; virtual;
    function MCopy: PObject; virtual;
    procedure ListError(aCode, aInfo: integer); virtual;
    function At(Index:integer): Pointer;
    function Last: Pointer;
    procedure Insert(aItem : Pointer); virtual;
    procedure AtInsert(aIndex : integer; aItem : Pointer); virtual;
    procedure InsertList(var aAnother : MList); virtual;
    function GetObject(aIndex:integer): Pointer; virtual;
    function IndexOf(aItem : Pointer): integer; virtual;
    procedure DeleteAll; virtual;
    procedure FreeItem(Item : Pointer); virtual;
    procedure FreeAll; virtual;
    procedure FreeItemsFrom(aIndex : integer); virtual;
    procedure Pack; virtual;
    procedure SetLimit(ALimit: integer); virtual;
    procedure Append To (var fAnother: MList); virtual;
    procedure TransferItems(var fAnother : MList); virtual;
    procedure CopyItems(var fOrigin : MList); virtual;
  end;
```

§196 Mizar Parser MIZAR LIST 65

196.	Growth factor.	How quickly an	Dynamic Ar	ray grows i	is a subject	of debate.	Just for a table of
the growth factors:							

Implementation	Growth Factor			
Java's ArrayList	3/2 = 1.5			
Microsoft's Visual C++	3/2 = 1.5			
Facebook folly/FBVector	3/2 = 1.5			
Unreal Engine's TArray	$n + ((3n) \gg 3) \sim 1.375$			
Python PyListObject	$n + (n \gg 3) \sim 1.125$			
Go slices	between 1.25 and 2			
Gnu C++	2			
Clang	2			
Rust's Vec	2			
Nim sequences	2			
SBCL vectors	2			
C#	2			

The *MList* uses a staggered growth factor, specifically something like $s(n) \leftarrow s(n) + GrowLimit(s(n))$. The sequence of Dynamic Array size would be:

$$s(n) = (0, 4, 8, 12, 28, 44, 60, 76, \ldots)$$

followed by $s(n+1) \leftarrow (5/4)s(n)$. I am not sure this is optimal, but I have no better solution.

CAUTION: If the memory allocator uses a first-fit allocation, then growth factors like $\alpha \geq 2$ can cause dynamic array expansion to run out of memory even though a significant amount of memory may still be available. For a discussion about this point, see:

• http://www.gahcep.com/cpp-internals-stl-vector-part-1/

It seems that a growth factor $\alpha \leq \varphi = (1 + \sqrt{5})/2$ must be not bigger than the golden ratio. To see this, we need a dyanmic array of size S to have its first growth to allocate αS , then frees up the S bytes from the pre-growth allocation. The second allocation needs $\alpha^2 S$ bytes. Observe the first two allocations requires $S + \alpha S$ bytes available. Now suppose we want this to be able to fit into the newly freed space,

$$\alpha^2 S \leq S + \alpha S$$

which means

end:

$$\alpha^2 - \alpha + 1 < 0$$

or (requiring $\alpha > 0$)

$$\alpha \le \varphi = \frac{1 + \sqrt{5}}{2}.$$

When this fails to hold, a first-fit allocation could run out of memory.

```
\langle \mathit{MList} \text{ implementation } 196 \rangle \equiv \{ \mathit{Simple Collection} \} 
\mathbf{function} \ \mathit{GrowLimit}(\mathit{aLimit} : \mathit{integer}) : \mathit{integer}; 
\mathbf{begin} \ \mathit{GrowLimit} \leftarrow 4; 
\mathbf{if} \ \mathit{aLimit} > 64 \ \mathbf{then} \ \mathit{GrowLimit} \leftarrow \mathit{aLimit} \ \mathbf{div} \ 4
```

This code is used in section 183.

197. Constructor. The constructor creates an empty list.

```
constructor MList.Init(aLimit: integer);
```

else if aLimit > 8 then $GrowLimit \leftarrow 16$;

begin $MObject.Init; Items \leftarrow nil; Count \leftarrow 0; Limit \leftarrow 0; SetLimit(aLimit); end;$

66 MIZAR LIST Mizar Parser §198

198. Moving a list into the caller.

```
constructor MList.MoveList(\mathbf{var}\ aAnother: MList);

begin MObject.Init;

Count \leftarrow aAnother.Count;\ Limit \leftarrow aAnother.Limit;\ Items \leftarrow aAnother.Items;\ \{ move \}

aAnother.DeleteAll;\ aAnother.Limit \leftarrow 0;\ aAnother.Items \leftarrow \mathbf{nil};\ \{ delete\ aAnother \}

end;
```

199. Copying the contents of a Another list into the current list will essentially reinitialize the current list, the insert all items from the other list into the current list.

```
constructor MList.CopyList(\mathbf{var}\ aAnother: MList);

begin MObject.Init;\ Items \leftarrow \mathbf{nil};\ Count \leftarrow 0;\ Limit \leftarrow 0;\ \{initialize\}

SetLimit(aAnother.Limit);\ InsertList(aAnother);

end;
```

200. A list is "done" frees all items in the list, sets the limit to zero, and then invokes the superclass's *Done* method.

```
destructor MList.Done;
begin FreeAll; SetLimit(0); inherited Done; end;
```

201. We override the MObject.MCopy method (§190). This will copy the base object using CopyObject (§189), allocate a new array of pointers, copy over the contents of the caller, and then returns the new list.

```
function MList.MCopy: PObject;

var lList: PObject; i: integer;

begin lList \leftarrow CopyObject; GetMem(PList(lList)\uparrow.Items, Self.Limit * SizeOf(Pointer));

for i \leftarrow 0 to Self.Count - 1 do PList(lList)\uparrow.Items\uparrow[i] \leftarrow PObject(Self.Items\uparrow[i])\uparrow.MCopy;

MCopy \leftarrow lList;

end;
```

202. This is the same as MList.GetObject (§207), and I am not sure why we have two versions of the same function.

```
function MList.At(Index:integer): Pointer;
begin if (Index < 0) \lor (Index \ge Count) then
begin ListError(coIndexError, 0); At \leftarrow nil; end
else At \leftarrow Items \uparrow [Index];
end;
```

203. The MList.Count tracks the number of allocated items. So the last item would be located at MList.Count - 1 (since we count with zero offset).

```
function MList.Last: Pointer;
begin Last \leftarrow At(Count - 1); end;
```

204. Inserting an item into a list requires checking there's enough free space to the list, then sets the first spot to the item.

```
procedure MList.Insert(aItem : Pointer);

begin if Limit = Count then SetLimit(Limit + GrowLimit(Limit));

Items \uparrow [Count] \leftarrow aItem; inc(Count);

end;
```

§205 Mizar Parser MIZAR LIST 67

205. If we want to insert a pointer at a specific index, then we proceed as follows:

- (1) Check if the index is negative. If so, then we should flag an error using ListError, and exit.
- (2) Check if the index is larger than the logical size of the dynamic array; if so, then we grow the dynamic array using SetLimit

```
procedure MList.AtInsert(aIndex:integer; aItem:Pointer);
var i, lLimit:integer;
begin if aIndex < 0 then
begin ListError(coIndexError, 0); exit;
end;
if (aIndex \ge Limit) \lor ((aIndex = Count) \land (Limit = Count)) then {ensure capacity}
begin lLimit \leftarrow Limit + GrowLimit(Limit);
while aIndex + 1 > lLimit do lLimit \leftarrow lLimit + GrowLimit(lLimit);
SetLimit(lLimit); {Copy contents}
end;
for i \leftarrow Count to aIndex - 1 do Items \uparrow [i] \leftarrow nil; {fill new entries as nil}
Items \uparrow [aIndex] \leftarrow aItem; {set the entry at aIndex to the pointer}
if aIndex \ge Count then Count \leftarrow aIndex + 1; {update logical size, if necessary}
end:
```

206. When we insert aAnother list into the current list, we simply iterate through all the other list's items, and insert (a copy of the pointer to) each one into the current list. This should leave aAnother list unmodified.

```
procedure MList.InsertList(\mathbf{var}\ aAnother: MList);

\mathbf{var}\ i:\ integer;

\mathbf{begin}\ \mathbf{for}\ i \leftarrow 0\ \mathbf{to}\ pred(aAnother.Count)\ \mathbf{do}\ Insert(PObject(aAnother.Items\uparrow[i])\uparrow.MCopy);

\mathbf{end};
```

207. Given an index, find the item located there. Well, the pointer to the object. When the index is illegal (out of bounds or negative), then flag an error and return **nil**. Otherwise return the pointer located at the index.

```
function MList.GetObject(aIndex:integer): Pointer;
begin if (aIndex < 0) \lor (aIndex \ge Count) then
begin ListError(coIndexError, 0); GetObject \leftarrow nil; end
else GetObject \leftarrow Items \uparrow [aIndex];
end;
```

208. We have a default error code for lists.

```
procedure MList.ListError(aCode, aInfo : integer);
begin RunError(212 - aCode); end;
```

209. Looking for the index of an item requires iterating through each item of the list, until we find the needle in the hay stack. Once found, we return the index for the needle.

If the needle is not in the haystack, return -1.

```
function MList.IndexOf (aItem:Pointer): integer; var i:integer; begin IndexOf \leftarrow -1; for i \leftarrow 0 to pred (Count) do
   if aItem = Items \uparrow [i] then
   begin IndexOf \leftarrow i; break end
end;
```

68 MIZAR LIST Mizar Parser $\S 210$

210. Deleting all items from a list simply updates the list's logical size (i.e., *Count*) to zero. This will not alter the underlying array allocated for the dynamic array.

```
procedure MList.DeleteAll; begin Count \leftarrow 0; end;
```

211. Freeing a single item will invoke PASCAL's primitive Dispose function (which frees up the memory in heap). This is a helper function to avoid accidentally invoking $Dispose(PObject(\mathbf{nil}), Done)$ which would throw errors.

```
procedure MList.FreeItem(Item : Pointer);
begin if Item ≠ nil then Dispose(PObject(Item), Done);
end;
```

212. We delegate all the heavy work of *FreeAll* to *FreeItemsFrom*.

```
 \begin{array}{c} \mathbf{procedure} \ \mathit{MList.FreeAll}; \\ \mathbf{begin} \ \mathit{FreeItemsFrom}(0); \ \mathbf{end}; \end{array}
```

213. We can itereate through a list from a start index, freeing the rest of the list starting from aIndex. Remember, the data structure for MList consists of an MObject extended with its capacity, logical size, and a pointer to the array on the heap. When freeing an item from the array, we dereference the pointer to look up item I in the array.

```
procedure MList.FreeItemsFrom(aIndex:integer);
var I: integer;
begin for I \leftarrow Count - 1 downto aIndex do FreeItem(Items\uparrow[I]);
Count \leftarrow aIndex;
end:
```

214. If an item has become **nil** in the list, we should shift the rest of the list down. Basically, in Lisp, if null (cadr 1), then setf 1 (cdr 1).

Care must be taken to iterate over the items in the list. Shifting items down by one item requires iterating over k from i to Count - 2 (because the maximum index is Count - 1 due to zero offset indexing).

Once we have shifted everything down, we decrement the logical size of the dynamic array.

```
procedure MList.Pack;

var i,k:integer;

begin for i \leftarrow Count - 1 downto 0 do

if Items \uparrow [i] = \mathbf{nil} then

begin for k \leftarrow i to Count - 2 do Items \uparrow [k] \leftarrow Items \uparrow [k+1];

dec(Count);

end;

end;
```

§215 Mizar Parser MIZAR LIST 69

- **215.** Growing a list handles a few edgecases:
- (1) If the new limit is *smaller* than the existing limit, then just set the new limit equal to the existing limit.
- (2) If the new limit is *larger* than the maximum limit, then just set the new limit equal to the maximum limit.
- (3) If the new limit is not equal to the existing limit, then we have the "standard situation".
 - (i) When the new limit is zero, simply set the pointer to the item list to nil
 - (ii) Otherwise (for a new limit which is a nonzero number), allocate a new chunk of memory for the number of pointers needed, then move them. Be sure to free up the pointers, and update the variables.

```
procedure MList.SetLimit(ALimit:integer);
var lItems: PItemList;
begin if ALimit < Count then ALimit \leftarrow Count;
if ALimit > MaxCollectionSize then ALimit \leftarrow MaxCollectionSize;
if ALimit \neq Limit then
begin if ALimit = 0 then lItems \leftarrow nil
else begin GetMem(lItems, ALimit * SizeOf(Pointer));
if ((Count) \neq 0) \land (Items \neq nil) then Move(Items \uparrow, lItems \uparrow, Count * SizeOf(Pointer));
end;
if Limit \neq 0 then FreeMem(Items, Limit * SizeOf(Pointer));
Items \leftarrow lItems; Limit \leftarrow ALimit;
end;
end;
```

216. Appending another list to the current list will expand the current list to support the new items, insert the other list's items at the end of the current list, and then free the other list from memory.

```
procedure MList.AppendTo(var\ fAnother: MList);

var k: integer;

begin SetLimit(Count + fAnother.Count);

for k \leftarrow 0 to fAnother.Count - 1 do Insert(fAnother.Items\uparrow[k]);

fAnother.DeleteAll; fAnother.Done;

end;
```

217. There is a comment in Polish at the beginning of this function stating "Przeznaczeniem tej procedury jest uzycie jej w konstruktorach Move, ktore wykonuja jakgdyby pelna instrukcje przypisania (razem z VMTP)" which Google translates as "The purpose of this procedure is to be used in Move constructors, which execute a full assignment statement (including VMTP)."

There is also another comment in Polish, "Nie wolno uzyc SetLimit, bo rozdysponuje Items" which I translated into English and kept inline ("You cannot use SetLimit because it will distribute the Items").

The semantics of $Object \leftarrow Object$ will copy the right-hand side to the left-hand side.

```
procedure MList.TransferItems(\mathbf{var}\ fAnother: MList);
begin Self \leftarrow fAnother; { copy contents of fAnother over to Self }
fAnother.DeleteAll; fAnother.Limit \leftarrow 0; fAnother.Items \leftarrow \mathbf{nil};
{ You cannot use SetLimit because it will distribute the Items. }
end;
```

218. Copying items from a list simply loops through the original list, inserting them into the caller.

```
procedure MList.CopyItems(var\ fOrigin: MList);

var i:\ integer;

begin for i \leftarrow 0 to fOrigin.Count - 1 do Insert(PObject(fOrigin.Items \uparrow [i]) \uparrow.CopyObject);

end;
```

Mizar Parser

Section 9.4. MIZAR COLLECTION CLASS

growth rate from s(n+1) = s(n) + GrowLimit(s(n)) to be

70

Curiously, the "Collection" class extends the "List" class, which surprises me. This will change the

```
s(n+1) = s(n) + GrowLimit(\Delta + s(n))
```

where $\Delta \geq 0$ is a field of the Collection object. When we move an MList into an MCollection, we have $Delta \leftarrow 2$ be the default value.

```
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
    { MCollection object }
  PCollection = \uparrow MCollection;
  MCollection = \mathbf{object} (MList)
    Delta: integer;
    constructor Init(ALimit, ADelta : integer);
    destructor Done; virtual;
    procedure AtDelete(Index: integer);
    procedure AtFree(Index : integer);
    procedure AtInsert(Index : integer; Item : Pointer); virtual;
    procedure AtPut(Index: integer; Item: Pointer);
    procedure Delete(Item : Pointer);
    procedure Free(Item : Pointer);
    procedure Insert(aItem : Pointer); virtual;
    procedure Pack; virtual;
    constructor MoveCollection(var fAnother : MCollection);
    constructor MoveList(var aAnother : MList);
    constructor CopyList(var aAnother : MList);
    constructor CopyCollection(var AAnother : MCollection);
    constructor Singleton(fSing : PObject; fDelta : integer);
    procedure Prune; virtual;
  end;
220. Constructor. When constructing a new Collection, we allocate an array of the desired limit (using
the SetLimit (§215) to handle this allocation).
\langle MCollection \text{ implementation } 220 \rangle \equiv
```

```
{ MCollection }
constructor MCollection. Init(ALimit, ADelta: integer);
  begin MObject.Init; Items \leftarrow \mathbf{nil}; Count \leftarrow 0; Limit \leftarrow 0; Delta \leftarrow ADelta; SetLimit(ALimit);
  end;
destructor MCollection.Done;
  begin FreeAll; SetLimit(0);
  end;
This code is used in section 183.
```

221. When trying to delete an element at *Index*, we first check if the *Index* is within the bounds of the collection. If it's out of bounds, we invoke *ListError* and exit the function.

Otherwise, we shift everything in the collection down by one position.

```
procedure MCollection.AtDelete(Index:integer);

var i: integer;

begin if (Index < 0) \lor (Index \ge Count) then

begin ListError(coIndexError, 0); exit; end;

if Index < pred(Count) then

for i \leftarrow Index to Count - 2 do Items \uparrow [i] \leftarrow Items \uparrow [i+1];

Dec(Count);

end;
```

222. If we want to also *free* an object in a collection, we store it in a temporary variable, then invoke AtDelete(Index) to update the collection, and finally *Free* the item.

```
procedure MCollection.AtFree(Index:integer);
var lItem: Pointer;
begin lItem \leftarrow At(Index); AtDelete(Index); FreeItem(lItem); end;
```

223. Inserting an item at an *Index*, we first need to check if the position is within the bounds of the collection. If it's out of bounds, then flag a *ListError* and exit the function.

Otherwise, we check if the collection is at capacity (Limit = Count). If so, we try to expand the collection by Delta items. When Delta is zero, then raise an error and exit.

Now we are at the "default" case. Simply shift items starting at *Index* up by one. Then set the item at *Index* to be the new *Item*, and increment the count of the collection.

```
procedure MCollection.AtInsert(Index:integer; Item:Pointer);
begin if (Index < 0) \lor (Index > Count) then
begin ListError(coIndexError, 0); exit; end;
if Limit = Count then
begin if Delta = 0 then
begin ListError(coOverFlow, 0); exit; end;
SetLimit(Limit + Delta);
end;
if Index \neq Count then Move(Items\uparrow[Index], Items\uparrow[Index + 1], (Count - Index) * SizeOf(pointer));
Items\uparrow[Index] \leftarrow Item; inc(Count);
end;
```

224. Overwrite contents at index. We can insert a new item at a given index without shifting the collection.

```
procedure MCollection.AtPut(Index:integer; Item:Pointer);
begin if (Index < 0) \lor (Index \ge Count) then ListError(coIndexError, 0)
else Items \uparrow [Index] \leftarrow Item;
end;
```

225. Deleting an item finds the index of the item, then invokes AtDelete on that index.

```
procedure MCollection.Delete(Item : Pointer);
begin AtDelete(IndexOf(Item)); end;
```

226. Similarly, freeing an item is just *Delete*-ing the item, then calling *FreeItem* on the pointer.

```
procedure MCollection.Free(Item : Pointer);
begin Delete(Item); FreeItem(Item); end;
```

72

```
227. Inserting an item at the end of the collection.
```

```
procedure MCollection.Insert(aItem : Pointer);
begin AtInsert(Count, aItem); end;
```

228. We can also "fit" the collection by deleting all nil elements.

```
procedure MCollection.Pack;
var i: integer;
begin for i \leftarrow pred(Count) downto 0 do
if Items \uparrow [i] = nil then AtDelete(i);
end;
```

229. Move semantics for creating a new collection.

```
constructor MCollection.MoveCollection(var fAnother : MCollection); begin Init(0, fAnother.Delta); TransferItems(fAnother) end;
```

230. Cloning a collection will simply create an empty collection, the loop through *AAnother* inserting each item from the original collection into the newly minted collection.

```
constructor MCollection.CopyCollection (var AAnother: MCollection);
var i: integer;
begin Init(AAnother.Limit, AAnother.Delta);
for i \leftarrow 0 to AAnother.Count - 1 do Insert(aAnother.Items \uparrow [i]);
end;
```

231. A singleton allocates as little as possible.

```
constructor MCollection.Singleton(fSing : PObject; fDelta : integer);
begin Init(2, fDelta); Insert(fSing) end;
```

232. Pruning a collection merely sets its limits to zero. It does not free the contents of the collection.

```
procedure MCollection.Prune;
begin SetLimit(0) end;
```

233. Moving an *MList* uses PASCAL's inheritance semantics to invoke *MList.MoveList* and then sets the *Delta* to 2.

```
constructor MCollection.MoveList(\mathbf{var}\ aAnother: MList); begin inherited\ MoveList(aAnother);\ Delta \leftarrow 2; end;
```

234. Copying a list invokes MList.CopyList on the collection, then sets $Delta \leftarrow 2$.

```
constructor MCollection.CopyList(\mathbf{var}\ aAnother: MList); begin inherited\ CopyList(aAnother);\ Delta \leftarrow 2; end;
```

Section 9.5. SIMPLE STACKED (EXTENDIBLE) LISTS

235. This is used to track newly registered clusters in Mizar.

The basic idea is that we partition the array into the first N entries, then the remaining k entries. The last k entries are the "extendible" entries.

We will eventually "digest" the extendible entries (by incrementing $N \leftarrow N+1$ and decrementing $k \leftarrow k-1$ until k=0).

```
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
    { MExtList object }
  MExtListPtr = \uparrow MExtList;
  MExtList = \mathbf{object} (MList)
    fExtCount: integer;
    constructor Init(aLimit : integer);
    destructor Done; virtual;
    procedure Insert(aItem : Pointer); virtual;
    procedure Mark(var aIndex : integer); virtual;
    procedure FreeItemsFrom(aIndex : integer); virtual;
    procedure DeleteAll; virtual;
    procedure FreeAll; virtual;
    procedure Pack; virtual;
    procedure InsertExt(AItem : Pointer); virtual;
    procedure SetLimit(ALimit : integer); virtual;
    procedure AddExtObject; virtual;
    procedure AddExtItems; virtual;
    procedure DeleteExtItems;
    procedure FreeExtItems;
  end;
236. Simple Stacked (Extendable) Collection.
\langle MExtList \text{ implementation } 236 \rangle \equiv
constructor MExtList.Init(ALimit : integer);
  begin MObject.Init;
  Items \leftarrow \mathbf{nil};
  Count \leftarrow 0; Limit \leftarrow 0;
  SetLimit(ALimit); fExtCount \leftarrow 0;
  end;
This code is used in section 183.
```

237. Destructor for MExtList. The destructor for MExtList invokes self. Free ExtItems and then calls the inherited destructor from the superclass.

```
destructor MExtList.Done;
begin FreeExtItems; inheritedDone; end;
```

Mizar Parser

74

Inserting an item. The *fExtCounter* field is unclear to me. But if it's nonzero, then an error has occurred and we bail out.

Otherwise, we possibly grow the extendible list, and we insert at the end the given pointer and increment the Count of items allocated.

```
procedure MExtList.Insert(aItem: Pointer);
  begin if fExtCount \neq 0 then
    begin ListError(coIndexExtError, 0); exit; end;
  if Limit = Count then SetLimit(Limit + GrowLimit(Limit));
  Items \uparrow [Count] \leftarrow aItem; \{ Append the item to the list \}
  inc(Count);
  end;
```

239. **Deleting all entries.** We can only call this when the extendible entries have been "digested" into the underlying array (i.e., when fExtCount = 0). Otherwise we need to flag an error. Otherwise, when all the extendible entries have been "digested", we call the parent's DeleteAll method.

```
procedure MExtList.DeleteAll;
  begin if fExtCount \neq 0 then
    begin ListError(coIndexExtError, 0); exit;
    end:
  inherited\, Delete All\,;
  end;
```

Free all entries. Like deleting all the entries, we need to fully digest all the extendible entries before invoking the parent class's FreeAll method. If there are extendible entries not fully digested, then we get indigestion (i.e., a list error).

```
procedure MExtList.FreeAll;
  begin if fExtCount \neq 0 then
    begin ListError(coIndexExtError, 0); exit;
    end:
  inherited FreeAll;
  end;
```

241. Packing. When packing an extendible list, we assert the extendible items have been digested fully. If not, raise an error. If fully digested, then invoke the parent class's Pack method.

```
procedure MExtList.Pack;
  begin if fExtCount \neq 0 then
    begin ListError(coIndexExtError, 0); exit;
    end:
  inherited Pack;
  end:
```

Insert extendible items. We can add an extendible item by first growing the list (if necessary), then adding an item at index N+k. Then increment the number of extendible items $k \leftarrow k+1$.

```
procedure MExtList.InsertExt(AItem : Pointer);
  begin if Limit = Count + fExtCount then SetLimit(Limit + GrowLimit(Limit));
  Items \uparrow [Count + fExtCount] \leftarrow AItem; inc(fExtCount);
  end:
```

243. Ensure capacity of extendible list.

- (1) When the new limit is less than the logical size N and the extendible size k, we just set the capacity to N + k.
- (2) Else if the new limit is larger than MaxCollectionSize, then just use the maximum collection size as the capacity.
- (3) Else if the new limit is different than the existing capacity, then we have to check if the new limit is zero. When it is, just set the capacity to zero and the list of items to **nil**. Otherwise, allocate space for a new array, and move over the contents from the existing array (and then free the existing array). Update the capacity and pointer to the items.

```
procedure MExtList.SetLimit(ALimit:integer);
var lItems: PItemList;
begin if ALimit < Count + fExtCount then ALimit \leftarrow Count + fExtCount;
if ALimit > MaxCollectionSize then ALimit \leftarrow MaxCollectionSize;
if ALimit \neq Limit then
begin if ALimit = 0 then lItems \leftarrow nil
else begin GetMem(lItems, ALimit * SizeOf(Pointer));
if ((Count + fExtCount) \neq 0) \land (Items \neq nil) then
Move(Items\uparrow, lItems\uparrow, (Count + fExtCount) * SizeOf(Pointer));
end;
if Limit \neq 0 then FreeMem(Items, Limit * SizeOf(Pointer));
Items \leftarrow lItems; Limit \leftarrow ALimit;
end;
end;
```

244. "Marking" an extendible list amounts to setting the procedure's variable to the capacity of the extendible list.

```
procedure MExtList.Mark(\mathbf{var}\ aIndex: integer); begin aIndex \leftarrow Count; end;
```

245. Freeing items starting at a given index requires the extendible items to be fully digested (if not, raise an error). Then simply free each object using the virtual destructor *MObject.Done*.

```
procedure MExtList.FreeItemsFrom(aIndex:integer);

var I: integer;

begin if fExtCount \neq 0 then

begin ListError(coIndexExtError, 0); exit;

end;

for I \leftarrow Count - 1 downto aIndex do

if Items\uparrow[I] \neq nil then Dispose(PObject(Items\uparrow[I]), Done);

Count \leftarrow aIndex;

end;
```

246. Digesting one extendible item. We can instruct the extendible list to digest exactly one extendible item. This requires the number of extendible items to be positive k > 0. If not, raise an error. Otherwise increment the logical capacity $N \leftarrow N+1$ and decrement the number of extendible items $k \leftarrow k-1$.

```
procedure MExtList.AddExtObject;
begin if fExtCount \le 0 then
begin ListError(coIndexExtError, 0); exit;
end;
inc(Count); dec(fExtCount);
end;
```

76

247. Digest all extendible items. This simply updates capacity to be incremented by the number of extendible items. Then the number of extendible items is set to zero. No error is raised if there are no extendible items (unlike digesting one single extendible item).

```
procedure MExtList.AddExtItems; begin Count \leftarrow Count + fExtCount; fExtCount \leftarrow 0; end;
```

248. Deleting all extendible items simply sets the *number* of extendible items to zero. This is a "soft delete" which does not affect anything else on the heap.

```
procedure MExtList.DeleteExtItems; begin fExtCount \leftarrow 0; end;
```

249. Freeing all the extendible items will "hard delete" each extendible item, removing them from the heap.

```
procedure MExtList.FreeExtItems;
var I: integer;
begin for I \leftarrow 0 to fExtCount - 1 do
if Items\uparrow[Count + I] \neq nil then Dispose(PObject(Items\uparrow[Count + I]), Done);
fExtCount \leftarrow 0;
end;
```

§250 Mizar Parser SORTED LISTS 77

Section 9.6. SORTED LISTS

This code is used in section 183.

These are used in the equalizer and in the correlator, specifically for keeping a collection of identifiers. A sorted list uses an array of indices (called *fIndex*). The array of indices are sorted according to a comparison of values. Invariant: Length(fIndex) = Length(Items)Invariant (sorted): for each $i = 0, \dots, Length(Items) - 2$, we have $Items \uparrow [fIndex \uparrow [i]] \le Items \uparrow [fIndex \uparrow [i]]$ 1]]. Also, we are taking the convention that fCompare(x,y) returns -1 when x < y; returns 0 when x = y; returns +1 when x > y. $\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv$ { MSortedList Object } $IndexListPtr = \uparrow MIndexList;$ MIndexList = array [0...MaxCollectionSize - 1] of integer;CompareProc =**function** (aItem1, aItem2: Pointer): integer; $MSortedList = \mathbf{object} (MList)$ fIndex: IndexListPtr; fCompare: CompareProc; constructor Init(ALimit : integer);constructor InitSorted(aLimit : integer; aCompare : CompareProc); **constructor** MoveList(var aAnother : MList); **constructor** CopyList(**const** aAnother: MList); **procedure** AtInsert(aIndex:integer; aItem:Pointer); virtual; **procedure** Insert(aItem : Pointer); virtual; **function** IndexOf(aItem: Pointer): integer; virtual; **procedure** Sort(aCompare : CompareProc); procedure SetLimit(ALimit : integer); virtual; **function** Find(aKey : Pointer; var aIndex : integer): boolean; virtual; **function** Search(aKey: Pointer; var aIndex: integer): boolean; virtual; **procedure** Pack: virtual: **procedure** FreeItemsFrom(aIndex:integer); virtual; end; **251.** Constructors. There are four constructors: (1) *Init* simply creates an empty list with a given capacity. (2) InitSorted is like Init, but expects an ordering operator. (3) MoveList moves all the items from another list into the caller, sorting as needed. This will also empty the other list. (4) CopyList is like MoveList but leaves the other list untouched. $\langle MSortedList \text{ implementation } 251 \rangle \equiv$ { MSortedList object } **constructor** MSortedList.Init(aLimit:integer); **begin** $MObject.Init;\ Items \leftarrow \mathbf{nil};\ Count \leftarrow 0;\ Limit \leftarrow 0;\ fIndex \leftarrow \mathbf{nil};\ fCompare \leftarrow \mathbf{nil};$ SetLimit(ALimit);end; **constructor** MSortedList.InitSorted(aLimit:integer; aCompare:CompareProc); **begin** $MObject.Init; Items \leftarrow nil; Count \leftarrow 0; Limit \leftarrow 0; fIndex \leftarrow nil; fCompare \leftarrow aCompare;$ SetLimit(ALimit);end; See also section 263.

78 SORTED LISTS Mizar Parser $\S 252$

252. Move constructor. When we move items from an MList into the caller, we also sort as we insert constructor $MSortedList.MoveList(var\ aAnother: MList)$;

```
var I: integer;

begin Items \leftarrow aAnother.Items; Count \leftarrow aAnother.Count; Limit \leftarrow aAnother.Limit;

GetMem(fIndex, Limit * SizeOf (integer)); fCompare \leftarrow nil;

for I \leftarrow 0 to pred (aAnother.Count) do fIndex\uparrow[I] \leftarrow I;

{ Empty out the other list }

aAnother.DeleteAll; aAnother.Limit \leftarrow 0; aAnother.Items \leftarrow nil;

end;
```

253. The *CopyList* constructor is like the *MoveList* **except** that the other list is not modified.

```
constructor MSortedList.CopyList (const aAnother: MList);

var i: integer;

begin MObject.Init; Items \leftarrow \mathbf{nil}; Count \leftarrow 0; Limit \leftarrow 0; fIndex \leftarrow \mathbf{nil}; fCompare \leftarrow \mathbf{nil};

SetLimit(aAnother.Limit); Count \leftarrow aAnother.Count;

for i \leftarrow 0 to Count - 1 do

begin Items\uparrow[i] \leftarrow PObject(aAnother.Items\uparrow[i])\uparrow.MCopy; fIndex\uparrow[I] \leftarrow I;

end;

end;
```

254. Insert element at an index. We can insert (potentially overwriting an existing entry) at a given index.

```
{ used in CollectCluster not to repeat the search, should be used only when @fCompare \neq nil } procedure MSortedList.AtInsert(aIndex:integer; aItem:Pointer); begin if Limit = Count then SetLimit(Limit + GrowLimit(Limit)); { Ensure capacity } if aIndex \neq Count then Move(fIndex\uparrow[aIndex], fIndex\uparrow[aIndex+1], (Count-aIndex)*SizeOf(integer)); Items\uparrow[Count] \leftarrow aItem; fIndex\uparrow[aIndex] \leftarrow Count; inc(Count); end;
```

- 255. Inserting an item. Inserting an item into a sorted list boils down to two cases:
- (1) If there is an ordering operator, we check if the item is in the underlying array using Find (§260), which will mutate the lIndex to be where it should be located. When the item is missing, simply insert it at lIndex. When the item is present, then we do nothing.
- (2) If there is no ordering operator, then check if the item already is present in the sorted list. If so, then don't do annything. Otherwise, insert the item at the start of the list.

```
procedure MSortedList.Insert(aItem:Pointer);

var lIndex:integer;

begin if @fCompare = nil then

begin if Limit = Count then SetLimit(Limit + GrowLimit(Limit));

Items \uparrow [Count] \leftarrow aItem; fIndex \uparrow [Count] \leftarrow Count; inc(Count); exit;

end;

if \neg Find(aItem, lIndex) then AtInsert(lIndex, aItem);

end;
```

§256 Mizar Parser SORTED LISTS 79

256. Resizing a sorted list. The invariant is that the list is sorted when it has an ordering operator (and so restricting to *aLimit* preserves the list being sorted), and it is a "set" when it does not have an ordering (and so restricting to *aLimit* preserves this property of being a finite set without duplicate entries).

```
procedure MSortedList.SetLimit(aLimit:integer);
  var lItems: PItemList; lIndex: IndexListPtr;
  begin if aLimit < Count then aLimit \leftarrow Count;
  if aLimit > MaxCollectionSize then aLimit \leftarrow MaxCollectionSize;
  if aLimit \neq Limit then
    begin if aLimit = 0 then
       begin lItems \leftarrow nil; lIndex \leftarrow nil; end
    else begin GetMem(lItems, aLimit * SizeOf(Pointer)); GetMem(lIndex, aLimit * SizeOf(integer));
       if Count \neq 0 then
         begin if Items \neq nil then
            begin Move(Items \uparrow, IItems \uparrow, Count * SizeOf(Pointer));
            Move(fIndex\uparrow, IIndex\uparrow, Count * SizeOf(integer));
         end;
       end;
    if Limit \neq 0 then
       begin FreeMem(Items, Limit * SizeOf(Pointer)); FreeMem(fIndex, Limit * SizeOf(integer));
    Items \leftarrow lItems; \ fIndex \leftarrow lIndex; \ Limit \leftarrow aLimit;
  end;
```

80 SORTED LISTS Mizar Parser $\S 257$

257. Quick sort an array. We have a private helper function for quicksorting an IndexListPtr (§250). Initially $L \leftarrow 0$ and $R \leftarrow length(aList) - 1$. It may be instructive to compare this to Algorithm Q in $The\ Art$ of $Computer\ Programming$, third ed., volume 3, §5.2.2. Specifically Mizar appears to use Hoare partitioning. We can summarize its algorithm thus:

- **Algorithm S** (*Quicksort*). This uses Hoare partition. We assume that $L \leq R$, and that *aCompare* is a total order (it's transitive an the law of trichotomy holds on all pairs of elements). Steps S1 through S4 are better known as the "partition" procedure.
- **S0.** [Initialize] Set $I \leftarrow L$, $J \leftarrow R$, and the pivot index $P_{idx} \leftarrow (L+R)$ shr 1, and the pivot value $P \leftarrow aList \uparrow [aIndex \uparrow [(L+R) \text{ shr 1}]]$. Observe $I \leq P_{idx} \leq J$ at this point.
- **S1.** [Move I right] While aList[I] < P, we increment $I \leftarrow I + 1$. This is guaranteed to terminate since $I \le P_{idx}$, so eventually we will get to aList[i] = P.
- **S2.** [Move J left] While P < aList[J], we decrement $J \leftarrow J 1$. This is guaranteed to terminate since $P_{idx} \leq J$, so eventually we will get to aList[J] = P.
- **S3.** [Keep going?] If I > J, then we're done "partitioning" (so everything to the left of the pivot is not greater than the pivot value, and everything to the right of the pivot is not lesser than the pivot value), and we go to step S5; otherwise go to the next step.
- **S4.** [Swap entries I and J] We swap the entries located at I and J, then set $I \leftarrow I+1$, and $J \leftarrow J-1$. If $I \leq J$, then return to step S1.
- **S5.** [Recur on left half] If L < J, then recursively call quicksort on the left half of the index (entries between $L \dots J 1$).
- **S6.** [Sort the right half] If $I \geq R$, then terminate. Otherwise, set $L \leftarrow I$ and return to step S0.

For readability, we also introduce a WEB macro for swapping the indices.

```
define steal\_from(\#) \equiv aIndex\uparrow[\#]; aIndex\uparrow[\#] \leftarrow T;
   define swap\_indices(\#) \equiv T \leftarrow aIndex\uparrow[\#]; aIndex\uparrow[\#] \leftarrow steal\_from
procedure ListQuickSort(aList: PItemList; aIndex: IndexListPtr; L, R: integer;
            aCompare : CompareProc);
   var I, J, T: integer; P: Pointer;
   begin repeat I \leftarrow L; J \leftarrow R; P \leftarrow aList \uparrow [aIndex \uparrow [(L+R) \text{ shr } 1]];
      repeat
         \{ I \le (L+R) \text{ shr } 1 \le J \}
         while aCompare(aList\uparrow [aIndex\uparrow [I]], P) < 0 do inc(I);
         \{ P \leq aList \uparrow [aIndex \uparrow [I]] \}
         while aCompare(aList\uparrow[aIndex\uparrow[J]], P) > 0 do Dec(J);
         \{ aList \uparrow [aIndex \uparrow [J]] \leq P \}
         \{ I \le (L+R) \text{ shr } 1 \le J \}
         if I \leq J then
           begin
            \{ \ aList \uparrow [aIndex \uparrow [J]] < P < aList \uparrow [aIndex \uparrow [I]] \}
            swap\_indices(I)(J);
            \{ aList \uparrow [aIndex \uparrow [I]] < P < aList \uparrow [aIndex \uparrow [J]] \} 
            \{I < J \text{ implies } inc(I) \le dec(J)\}\
            \{I = J \text{ implies } inc(I) > dec(J)\}
            inc(I); Dec(J);
            end;
      until I > J;
      \{J \leq (L+R) \text{ shr } 2 \leq I \text{ and } J < I\}
     if L < J then ListQuickSort(aList, aIndex, L, J, aCompare); {quicksort left half}
      L \leftarrow I; { recursively quicksort the right half of the array }
   until I \geq R;
   end;
```

258. Remarks.

- (1) It is unclear to me whether we must have a Compare be a linear order, and not a total pre-order. The difference is: do we really need $a \le b \land b \le a \implies a = b$ (i.e., a total order) or not (i.e., a total pre-order)?
- (2) PRECONDITION: We need to prove the *compare* operators are total orders for quicksort to work as expected.
- (3) ASSERT: Upon arriving to step Q5, the entries in L ... J 1 are partitioned (i.e., less than the pivot value) as is the entries in I ... R. In particular, the maximal element in L ... J 1 is located at J 1 while the minimal element in I ... R is located at I.
- (4) Robert Sedgewick's *Quicksort* (1980) is literally *the* book on the subject. An abbreviated reference may be found in Sedgewick's "The Analysis of Quicksort Programs" (*Acta Inform.* 7 (1977) 327–355, eprint)
- (5) IMPROVEMENT: This can be improved when recursively sorting the left half of the arrays by first checking if $J-L \leq 9$ then use insertion sort otherwise recursively quicksort the left half. (Similarly, instead of iterating the outermost while-loop, we should test if $R-I \leq 9$ then invoking insertion on the subarray indexed by I ... R.)
- (6) IMPROVEMENT: The pivot index P_{ind} is selected as $P_{ind} \leftarrow (L+R)/2$, which can lead to overflow. A safer way to compute this would be $P_{ind} \leftarrow L + ((R-L)/2)$.

According to the paper by Sedgewick we cited, when quicksorting a list of size less than M with a different sorting algorithm, the optimal choice of M (the cutoff for delegating to another sort algorithm) contributes to the runtime of quicksort,

$$f(M) = \frac{1}{6} \left(8M + 71 - 70H_{M+2} + \frac{270}{M+2} + \frac{54}{2M+3} - 36\frac{H_{M+1}}{M+2} \right).$$

We can use the approximation for Harmonic numbers

$$H_n = \ln(n) + \gamma + \frac{1}{2n} + O(n^{-2})$$

where $\gamma \approx 0.57721$ is Euler-Mascheroni constant. Using this replacement, we have

$$f'(M) \approx \frac{4}{3} + \frac{3}{(1+m)^2} - \frac{6}{1+m} + \frac{36\gamma - 253}{6(2+m)^2} - \frac{17}{3(2+m)} - \frac{18}{(3+2m)^2} + \frac{6\ln(1+m)}{(2+m)^2}.$$

We can numerically find the root for this to be $m_0 \approx 8.9888$ which gives a global minimum of $f(9) \approx -8.47671$.

This analysis is sketched out in Knuth's *The Art of Computer Programming*, volume III, but it may be worth sitting down and working this analysis out more fully.

259. Sorting a sorted list. We can update a sorted list to sort according to a new ordering operator, and also update the data structure to record this new ordering operator. This relies on ListQuickSort (§257) to do the actual sorting.

procedure MSortedList.Sort(aCompare : CompareProc);
var I: integer;

begin $fCompare \leftarrow aCompare;$

for $I \leftarrow 0$ to Count - 1 do $fIndex \uparrow [I] \leftarrow I$;

if (Count > 0) then ListQuickSort(Items, fIndex, 0, Count - 1, aCompare); end;

82 SORTED LISTS Mizar Parser §260

260. Find item. Finding an item in a sorted list boils down to two cases: do we have *fCompare* populated or not? If so, then use a binary search. If not, then just iterate item-by-item testing if *aKey* is in the underlying array.

CAUTION: The "find" function returns the index for the fIndex field, **NOT** the index for the underlying array of values (inherited from the MList class).

```
function MSortedList.Find(aKey:Pointer; var aIndex:integer): boolean; var <math>L, H, I, C: integer; begin Find \leftarrow False; if @fCompare = nil then \ \langle Find needle in <math>MSortedList by brute force 262\rangle; \langle Find needle in <math>MSortedList by binary search 261\rangle end;
```

261. Binary search is a little clever. We have L be the lower bounds index, and H the upper bounds index. The midpoint is obtained by taking their sum L + H and shifting to the right by 1 bit (which corresponds to dividing by 2, truncating the result).

We compare the item located at the midpoint to the given aKey, and store the result of this comparison in the variable C. If C < 0, then aKey is located to the right of the midpoint (so set $L \leftarrow I + 1$).

On the other hand, if $C \ge 0$, update $H \leftarrow I - 1$. When C = 0 (i.e., the midpoint is equal to aKey), then we set $L \leftarrow I + 1$ so we have H < L to terminate the loop. We set the return value to True when C = 0, and we mutate the aIndex to the index where we found the needle in the haystack.

```
\langle Find needle in MSortedList by binary search 261\rangle \equiv L \leftarrow 0; H \leftarrow Count - 1; while L \leq H do

begin I \leftarrow (L+H) shr 1; C \leftarrow fCompare(Items \uparrow [fIndex \uparrow [I]], aKey); if C < 0 then L \leftarrow I + 1

else begin H \leftarrow I - 1;

if C = 0 then

begin Find \leftarrow True; L \leftarrow I; end;

end;

end;

aIndex \leftarrow L;
```

262. We can simply iterate through the underlying array, testing item-by-item if each entry is equal to the needle or not.

```
\langle Find needle in MSortedList by brute force 262 \rangle \equiv begin aIndex \leftarrow Count; for I \leftarrow 0 to Count - 1 do
   if aKey = Items \uparrow [I] then
   begin Find \leftarrow True; \ aIndex \leftarrow I; \ break \ end; exit; end

This code is used in section 260.
```

This code is used in section 260.

§263 Mizar Parser SORTED LISTS 83

263. Search. We recall that *Find* returns the index of the *fIndex* field matching the needle. Usually, we want to know the index of the value itself. This is what *Search* performs.

```
\langle \mathit{MSortedList} \; \mathrm{implementation} \; 251 \rangle + \equiv function \mathit{MSortedList}.\mathit{Search}(\mathit{aKey} : \mathit{Pointer}; \; \mathbf{var} \; \mathit{aIndex} : \mathit{integer}) : \mathit{boolean}; var \mathit{I} : \; \mathit{integer}; begin \mathit{aIndex} \leftarrow \mathit{Count}; \; \mathit{Search} \leftarrow \mathit{false}; if \mathit{Find}(\mathit{aKey}, \mathit{I}) \; \mathbf{then} begin \mathit{Search} \leftarrow \mathit{true}; \; \mathit{aIndex} \leftarrow \mathit{fIndex} \uparrow [\mathit{I}]; end; end;
```

264. Index of a needle. Given a "needle", where in the haystack is it? Well, we require the ordering operator be non-nil for the sorted list — otherwise raise an error. Then using Find (§260), check if the entry is present. If it is, then return the index for the underlying array of values.

If the needle is not in the haystack, return -1.

```
function MSortedList.IndexOf(aItem:Pointer): integer;
var I: integer;
begin if @fCompare = nil then
begin ListError(coSortedListError, 0); exit;
end;
IndexOf \leftarrow -1;
if Find(aItem, I) then
begin \{if \ I < fCount \ then \}
IndexOf \leftarrow fIndex \uparrow [I];
end;
end;
```

265. Packing a sorted list. Use the superclass's Pack method. Then, when there is an ordering operator present, sort the list.

```
procedure MSortedList.Pack;
var lCount: integer;
begin lCount \leftarrow Count; inheritedPack;
if (@fCompare \neq nil) \land (lCount > Count) then Sort(fCompare);
end;
```

84 SORTED LISTS Mizar Parser $\S 266$

266. Free items starting at an index. When we want to remove all items starting at index a, we simply iterate through the array of indices starting at entry i = a and delete the value associated with Items[i] when it is non-nil.

This will also keep the indices for the non-deleted entries.

```
procedure MSortedList.FreeItemsFrom(aIndex:integer);
var I,k:integer;
begin if aIndex = Count then exit;
{ Delete entries from the array of values }
for I \leftarrow aIndex to Count - 1 do
    if Items \uparrow [I] \neq nil then Dispose(PObject(Items \uparrow [I]), Done);
{ Update the array of indices }
k \leftarrow 0;
for I \leftarrow 0 to Count - 1 do
    begin if fIndex \uparrow [I] < aIndex then
    begin fIndex \uparrow [k] \leftarrow fIndex \uparrow [l]; inc(k); end;
    end;
if k \neq aIndex then ListError(coSortedListError, 0);
Count \leftarrow aIndex;
end;
```

Section 9.7. SORTED EXTENDIBLE LISTS

267. We want to handle a sorted (§250) version of extendible lists — an *MSortedExtList*. It's used in the correlator for functorial registrations and inferred definition constants.

Like MSortedList, we add a field fIndex for the indices of the entries. This will track the digested items, not the extendible items.

An important invariant: the ordering operator (fCompare) must be non-nil.

```
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
  MSortedExtList = \mathbf{object} \ (MExtList)
    fIndex: IndexListPtr;
    fCompare: CompareProc;
    constructor Init(ALimit : integer);
    constructor InitSorted(aLimit: integer; aCompare: CompareProc);
    destructor Done: virtual:
    function Find(aKey : Pointer; var aIndex : integer): boolean; virtual;
    function FindRight(aKey : Pointer; var aIndex : integer): boolean; virtual;
    function FindInterval(aKey: Pointer; var aLeft, aRight: integer): boolean; virtual;
    function AtIndex(aIndex:integer): Pointer; virtual;
    procedure Insert(aItem : Pointer); virtual;
    procedure Pack; virtual;
    procedure InsertExt(AItem : Pointer); virtual;
    procedure SetLimit(ALimit : integer); virtual;
    procedure FreeItemsFrom(aIndex : integer); virtual;
    procedure AddExtObject; virtual;
    procedure AddExtItems; virtual;
  end;
```

268. Constructors. The *Init* constructor should not be used, and should raise an error if anyone tries to use it.

Instead, the *InitSorted* should be used to construct a new [empty] sorted extendible list with a given ordering operator.

```
⟨ MSortedExtList implementation 268⟩ ≡
    { MSortedExtList always with possible duplicate keys, always sorted }
constructor MSortedExtList.Init(ALimit: integer);
begin ListError(coIndexExtError, 0); end;
constructor MSortedExtList.InitSorted(aLimit: integer; aCompare: CompareProc);
begin inheritedInit(aLimit); fCompare ← aCompare;
end;
This code is used in section 183.
```

269. destructor The destructor for sorted extendible lists is just the inherited destructor from extendible lists.

```
destructor MSortedExtList.Done;
begin inherited Done; end;
```

§270

270. Finding a needle in the haystack. We require *fCompare* to be non-nil and enforce that invariant by raising an error when it is nil.

Then we just use bisection search to find the needle in the haystack. Once found, we mutate aIndex to the index L of the fIndex array which indexes the needle.

```
{ find the left-most if duplicates } function MSortedExtList.Find(aKey:Pointer; var aIndex:integer): boolean; var <math>L, H, I, C: integer; begin if \neg Assigned(fCompare) then ListError(coIndexExtError, 0); Find \leftarrow False; L \leftarrow 0; H \leftarrow Count - 1; while L \leq H do begin I \leftarrow (L + H) \sinh 1; C \leftarrow fCompare(Items \uparrow [fIndex \uparrow [I]], aKey); if C < 0 then L \leftarrow I + 1 else begin H \leftarrow I - 1; if C = 0 then Find \leftarrow True; end; end; aIndex \leftarrow L; end;
```

271. Find the rightmost index for a needle in the haystack. Since the underlying array is sorted, we check to see if the needle is in the haystack. If it is, we keep incrementing *aIndex* until it is no longer indexing the needle.

So upon return, if it returns True, then the aIndex parameter is mutated to equal the rightmost index for the needle's appearance in the haystack.

```
{ find the left-most with higher aKey, this is where we can insert } function MSortedExtList.FindRight(aKey:Pointer; var aIndex:integer): boolean; begin if <math>Find(aKey,aIndex) then begin while (aIndex < Count) \land (0 = fCompare(Items \uparrow [fIndex \uparrow [aIndex]], aKey)) do inc(aIndex); FindRight \leftarrow true; end else FindRight \leftarrow false; end;
```

272. Since we allow duplicate values in a sorted extendible list, we will sometimes wish to know the "interval" of entries equal to a needle. This will mutate *aLeft* and *aRight* to point to the beginning and end of the interval.

When the needle is not in the haystack, the function will mutate the variables to ensure aRight < aLeft to stress the point.

Possible bug: This assumes the MSortedExtList.Find returns the left-most index where the needle appears in the haystack.

```
{ find the interval of equal guys } function MSortedExtList.FindInterval(aKey: Pointer; \mathbf{var}\ aLeft, aRight: integer): boolean; begin if <math>Find(aKey, aLeft) then begin aRight \leftarrow aLeft + 1; while (aRight < Count) \land (0 = fCompare(Items \uparrow [fIndex \uparrow [aRight]], aKey)) do inc(aRight); dec(aRight); FindInterval \leftarrow true; end else begin aRight \leftarrow aLeft - 1; FindInterval \leftarrow false; end; end;
```

273. Get value at index. We check if the index i is within bounds of the sorted extendible list. If not, then we raise an error.

Otherwise, the default course of action, we simply lookup the entry fIndex[i] and then lookup the entry in the array of values located with that index.

```
function MSortedExtList.AtIndex(aIndex:integer): Pointer; begin if (aIndex < 0) \lor (aIndex \ge Count) then ListError(coIndexExtError, 0); AtIndex \leftarrow Items \uparrow [fIndex \uparrow [aIndex]]; end;
```

274. Inserting items. We can only insert an item into an extendible list when it has fully digested all its extendible items. This requirement carries over to sorted extendible lists.

When there are no extendible items, we delegate the work to *InsertExt*.

```
procedure MSortedExtList.Insert(aItem : Pointer);
begin if fExtCount ≠ 0 then ListError(coIndexExtError, 0);
InsertExt(aItem); AddExtObject;
end;
```

275. Packing a sorted extendible list is unsupported, so just raise an error if anyone tries to use it.

```
procedure MSortedExtList.Pack;
  begin ListError(coIndexExtError, 0);
  end;
```

276. Adding an extendible item. We ensure there is sufficient capacity in the underlying array of items, then add *AItem* at the position located by the logical size of the array of items. We also increment the number of extendible items.

```
procedure MSortedExtList.InsertExt(AItem : Pointer);
begin if Limit = Count + fExtCount then SetLimit(Limit + GrowLimit(Limit));
Items\uparrow[Count + fExtCount] \leftarrow AItem; inc(fExtCount);
end:
```

Mizar Parser §277

277. Ensure capacity. We can ensure the capacity of a sorted extendible list to be at least as large as *ALimit*.

When ALimit is smaller than the current capacity of the sorted extendible list, we allocate new arrays and copy over the old data. More importantly: we keep the last fExtCount items as ("undigested") extendible items.

```
procedure MSortedExtList.SetLimit(ALimit: integer);
  var lItems: PItemList; lIndex: IndexListPtr;
  begin Count \leftarrow Count + fExtCount;
  if aLimit < Count then aLimit \leftarrow Count;
  if aLimit > MaxCollectionSize then aLimit \leftarrow MaxCollectionSize;
  if aLimit \neq Limit then
    begin if aLimit = 0 then
       begin lItems \leftarrow nil; \ lIndex \leftarrow nil;
       end
                   { Allocate new arrays for indices and items }
    else begin
       GetMem(lItems, aLimit * SizeOf(Pointer)); GetMem(lIndex, aLimit * SizeOf(integer));
       if Count \neq 0 then { Copy items and indices from old arrays to new ones }
         begin if Items \neq nil then
            begin Move(Items\uparrow, IItems\uparrow, Count * SizeOf(Pointer));
            Move(fIndex\uparrow, IIndex\uparrow, Count * SizeOf(integer));
            end;
         end;
       end;
    if Limit \neq 0 then { Free old arrays }
       begin FreeMem(Items, Limit * SizeOf(Pointer)); FreeMem(fIndex, Limit * SizeOf(integer));
    Items \leftarrow IItems; fIndex \leftarrow IIndex; Limit \leftarrow aLimit; {Update the caller to use new arrays}
  Count \leftarrow Count - fExtCount;
  end;
```

278. Freeing items starting at an index. We have two exceptional situations:

- (1) The fExtCount must be zero, and if it is nonzero, then an error is raised; and
- (2) If the index given is equal to the logical size of the sorted extendible list, then we terminate early (since there is nothing to do).

```
procedure MSortedExtList.FreeItemsFrom(aIndex: integer);
  var I, k: integer;
  begin if fExtCount \neq 0 then ListError(coIndexExtError, 0);
  if aIndex = Count then exit;
  { Free items indexed by I \geq aIndex }
  for I \leftarrow aIndex to Count - 1 do
     if Items\uparrow[I] \neq nil then Dispose(PObject(Items\uparrow[I]), Done);
  { Sort fIndex for entries less than aIndex }
  k \leftarrow 0;
  for I \leftarrow 0 to Count - 1 do
     begin if fIndex \uparrow [I] < aIndex then
       begin fIndex \uparrow [k] \leftarrow fIndex \uparrow [I]; inc(k);
       end;
     end:
  if k \neq aIndex then ListError(coSortedListError, 0);
  Count \leftarrow aIndex;
  end;
```

279. Digest an extendible object. When there are extendible objects left to digest among the values (i.e., when fExtCount > 0), When $fExtCount \le 0$, then raise an error (there's nothing left to digest).

The first extendible item left to be digested is located at *Count* in the array of items. Then we find the right most index for the same extendible item. We digest all of them at once, shifting the *fIndex* as needed. Note that the need to shift *fIndex* down by 1 is needed to keep the array of items sorted.

```
{\bf procedure}\ {\it MSortedExtList.AddExtObject};
```

```
var lIndex: integer;

begin if fExtCount \leq 0 then ListError(coIndexExtError, 0);

FindRight(Items\uparrow[Count], lIndex);

if lIndex \neq Count then \{ shift\ fIndex\ to\ right\ by\ 1 \}

Move(fIndex\uparrow[lIndex], fIndex\uparrow[lIndex+1], (Count-lIndex)*SizeOf(integer));

fIndex\uparrow[lIndex] \leftarrow Count; \{ extendible\ item's\ index \}

inc(Count); Dec(fExtCount);

end;
```

280. Digest all extendible items. We can simply iterate through all the extendible items, digesting them one-by-one.

```
procedure MSortedExtList.AddExtItems;
begin while fExtCount > 0 do AddExtObject;
end;
```

Mizar Parser §281

Section 9.8. SORTED LIST OF STRINGS

```
This is used in the kernel to track directives, as well as makenv and accdict needs it.
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
  MSortedStrList = \mathbf{object} \ (MSortedList)
    constructor Init(ALimit : integer);
    function IndexOfStr(const aStr: String): integer; virtual;
    function ObjectOf(const aStr: String): PObject; virtual;
  end;
282. Pointer comparison. For strings, it is faster to use pointer comparison than lexicographic ordering.
Although pointer comparison is a total linear order, it may not produce intuitive comparisons.
\langle MSortedStrList \text{ implementation } 282 \rangle \equiv
    { MSortedStrList }
function CompareStringPtr(aKey1, aKey2: Pointer): integer;
  begin if PStr(aKey1)\uparrow .fStr < PStr(aKey2)\uparrow .fStr then CompareStringPtr \leftarrow -1
  else if PStr(aKey1)\uparrow fStr = PStr(aKey2)\uparrow fStr then CompareStringPtr \leftarrow 0
    else CompareStringPtr \leftarrow 1;
  end;
This code is used in section 183.
       Constructor. We just defer to the InitSorted constructor for sorted lists (\S 251).
  As an invariant, the fCompare ordering operator is always assumed to be set to the CompareStringPtr.
There is no other way to construct a sorted string list besides this constructor, which enforces this invariant.
constructor MSortedStrList.Init(ALimit : integer);
  begin InitSorted(ALimit, CompareStringPtr);
  end;
284. We can locate a string by Find-ing its entry in the fIndex array.
function MSortedStrList.IndexOfStr(const aStr: string): integer;
  var I: integer; lStringObj: MStrObj;
  begin IndexOfStr \leftarrow -1;
  if @fCompare = nil then { Invariant violation }
    begin ListError(coSortedListError, 0); exit;
    end;
  lStringObj.Init(aStr);
  if Find(@lStringObj, I) then
    begin if I < Count then IndexOfStr \leftarrow fIndex\uparrow[I];
    end;
  end;
285. We also can return the pointer to the object, if it is present in the sorted string list.
function MSortedStrList.ObjectOf(const aStr: string): PObject;
  var I: integer;
  begin ObjectOf \leftarrow nil; I \leftarrow IndexOfStr(aStr);
  if I \geq 0 then ObjectOf \leftarrow Items \uparrow [I];
  end;
```

§286 Mizar Parser SORTED COLLECTIONS 91

Section 9.9. SORTED COLLECTIONS

```
286.
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
    { MSortedCollection object }
  PSortedCollection = \uparrow MSortedCollection;
  MSortedCollection = object (MCollection)
    Duplicates: boolean;
    fCompare: CompareProc;
    constructor Init(ALimit, ADelta : integer);
    constructor InitSorted(ALimit, ADelta: integer; aCompare: CompareProc);
    function Compare (Key1, Key2: Pointer): integer; virtual;
    function IndexOf(aItem : Pointer): integer; virtual;
    procedure Insert(aItem : Pointer); virtual;
    procedure InsertD(Item: Pointer); virtual;
    function KeyOf (Item : Pointer): Pointer; virtual;
    function Search(Key: Pointer; var Index: integer): boolean; virtual;
  end;
287. Constructors. We can construct a sorted collection without an ordering operator, and we can
construct one with an ordering operator.
\langle MSortedCollection implementation 287 \rangle \equiv
    { MSortedCollection }
constructor MSortedCollection.Init(aLimit, aDelta: integer);
  begin inherited\ Init(ALimit, ADelta);\ Duplicates \leftarrow False;\ fCompare \leftarrow nil;
  end;
constructor MSortedCollection.InitSorted(aLimit, aDelta: integer; aCompare: CompareProc);
  begin inherited Init(ALimit, ADelta); Duplicates \leftarrow False; fCompare \leftarrow aCompare;
  end:
This code is used in section 183.
      Comparing entries. This will invoke Abstract1 (§183) when there is no ordering operator, which
itself raises an error 211.
  Otherwise, this just invokes fCompare on the two entries.
function MSortedCollection.Compare(Key1, Key2 : Pointer): integer;
  begin if @fCompare = nil then Abstract1;
  Compare \leftarrow fCompare(Key1, Key2);
  end:
289. Find the right-most index for an item in the collection. Searching (§293) for the KeyOf (§292). This
function corrects for the possible bug in Search (§293) which will return the index just to the left of an item.
function MSortedCollection.IndexOf(aItem: Pointer): integer;
  var I: integer;
  begin IndexOf \leftarrow -1;
  if Search(KeyOf(aItem), I) then
    begin if Duplicates then
       while (I < Count) \land (aItem \neq Items \uparrow [I]) do inc(I);
    if I < Count then IndexOf \leftarrow I;
    end;
  end;
```

92 SORTED COLLECTIONS Mizar Parser §290

290. Insert the item when it is not in the collection (or if duplicates are allowed). Otherwise do not mutate the caller.

```
procedure MSortedCollection.Insert(aItem: Pointer);
  var I: integer;
  begin if \neg Search(KeyOf(aItem), I) \lor Duplicates then <math>AtInsert(I, aItem);
  end;
291.
      Insert an item if it's not in the collection (or if there are duplicates allowed in the collection).
Otherwise, delete the item and do not mutate the caller.
procedure MSortedCollection.InsertD(Item: Pointer);
  var I: integer;
  begin if \neg Search(KeyOf(Item), I) \lor Duplicates then <math>AtInsert(I, Item)
  else Dispose(PObject(Item), Done);
  end:
       We treat the item itself as the key, so return the item. That is to say, this is the identity function.
  It does not mutate the caller.
function MSortedCollection.KeyOf(Item: Pointer): Pointer;
  begin KeyOf \leftarrow Item;
  end;
       Binary search. This is binary search through a sorted collection.
  If there are duplicates, this will return the left-most index.
function MSortedCollection.Search(Key: Pointer; var Index: integer): boolean;
  var L, H, I, C: integer;
  begin Search \leftarrow False; L \leftarrow 0; H \leftarrow Count - 1;
  while L \leq H do
    begin I \leftarrow (L+H) shr 1; C \leftarrow Compare(KeyOf(Items \uparrow [I]), Key);
    if C < 0 then L \leftarrow I + 1
    else begin H \leftarrow I - 1:
       if C = 0 then
         begin Search \leftarrow True;
         if \neg Duplicates then L \leftarrow I;
         end;
       end;
    end:
  Index \leftarrow L;
  end;
       Perform the lexicographic ordering of (x_1, y_1) against (x_2, y_2).
function CompareIntPairs(X1, Y1, X2, Y2: Longint): integer;
  var lRes: integer;
  begin lRes \leftarrow CompareInt(X1, X2):
  if lRes = 0 then lRes \leftarrow CompareInt(Y1, Y2);
```

 $CompareIntPairs \leftarrow lRes;$

end;

§295 Mizar Parser STRING COLLECTION 93

Section 9.10. STRING COLLECTION

```
295.
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
    { MStringCollection object }
  PStringCollection = \uparrow MStringCollection;
  MStringCollection = \mathbf{object} \ (MSortedCollection)
    function Compare(Key1, Key2 : Pointer): integer; virtual;
    procedure FreeItem(Item : Pointer); virtual;
  end;
    { UnsortedStringCollection }
  PUsortedStringCollection = \uparrow StringColl;
  StringColl = \mathbf{object} (MCollection)
    procedure FreeItem(Item : pointer); virtual;
  end;
       String ordering operator. We have the usual lexicograph ordering as an operator ordering.
\langle String collection implementation 296\rangle \equiv
    { MStringCollection }
function CompareStr(aStr1, aStr2 : string): integer;
  begin if aStr1 < aStr2 then CompareStr \leftarrow -1
  else if aStr1 = aStr2 then CompareStr \leftarrow 0
    else CompareStr \leftarrow 1;
  end:
This code is used in section 183.
297. We then have a convenience function to handle pointer dereferencing.
function MStringCollection.Compare(Key1, Key2: Pointer): integer;
  begin Compare \leftarrow Compare Str(PString(Key1)\uparrow, PString(Key2)\uparrow);
  end;
       Freeing items. We can free an item by simply freeing the string. This is the same for unsorted
string collections, too.
procedure MStringCollection.FreeItem(Item : Pointer);
  begin DisposeStr(Item);
  end;
{ UnsortedStringCollection }
procedure StringColl.FreeItem(Item: pointer);
  begin DisposeStr(Item);
  end;
```

94 INT COLLECTIONS Mizar Parser §299

Section 9.11. INT COLLECTIONS

```
The TIntItem is needed for the unifier and equalizer.
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
    { MIntCollection object }
  IntPair = \mathbf{record} \ X, Y: integer;
    end:
  IntPairItemPtr = \uparrow IntPairItem;
  IntPairItem = \mathbf{object} \ (MObject)
    fKey: IntPair;
    constructor Init(X, Y : integer);
  end;
  IntPtr = \uparrow integer;
  PIntItem = \uparrow TIntItem;
  TIntItem = \mathbf{object} \ (MObject)
    IntKey: integer;
    constructor Init(fInt : integer);
  end;
  PIntKeyCollection = \uparrow TIntKeyCollection;
  TIntKeyCollection = \mathbf{object} (MSortedCollection)
    function KeyOf (Item : pointer): pointer; virtual;
    function Compare (Key1, Key2: pointer): integer; virtual;
  end;
  IntPairKeyCollectionPtr = \uparrow IntPairKeyCollection;
  IntPairKeyCollection = \mathbf{object} \ (MSortedCollection)
    function Compare(Key1, Key2 : pointer): integer; virtual;
    function ObjectOf(X, Y : integer): IntPairItemPtr; virtual;
    function FirstThat(X:integer): IntPairItemPtr; virtual;
  end;
300.
       TIntItem constructor. This just copies the given integer over to the newly allocated TIntItem
object.
\langle MIntCollection implementation 300 \rangle \equiv
    { MIntCollection }
constructor TIntItem.Init(fInt:integer);
  begin IntKey \leftarrow fInt;
  end:
This code is used in section 183.
301. We use TIntItems as keys in a TIntKeyCollection.
function TIntKeyCollection.KeyOf(Item: pointer): pointer;
  begin KeyOf \leftarrow addr(PIntItem(Item)\uparrow.IntKey);
  end;
```

§302 Mizar Parser INT COLLECTIONS 95

```
302.
       Comparing items just looks at the integers referred by the pointers.
function TIntKeyCollection.Compare(Key1, Key2: pointer): integer;
  begin Compare \leftarrow 1;
  if IntPtr(Key1)\uparrow < IntPtr(Key2)\uparrow then
    begin Compare \leftarrow -1; exit
  if IntPtr(Key1)\uparrow = IntPtr(Key2)\uparrow then Compare \leftarrow 0;
  end;
303. Constructor for pairs of integers.
constructor IntPairItem.Init(X, Y : integer);
  begin fKey.X \leftarrow X; fKey.Y \leftarrow Y;
  end;
304. Comparing two keys in a collection indexed by IntPairs is done "in the obvious way".
function IntPairKeyCollection.Compare(Key1, Key2: pointer): integer;
  \mathbf{begin} \ \ Compare \leftarrow CompareIntPairs(IntPairItemPtr(Key1)\uparrow .fKey.X,IntPairItemPtr(Key1)\uparrow .fKey.Y,
                                           IntPairItemPtr(Key2)\uparrow.fKey.X, IntPairItemPtr(Key2)\uparrow.fKey.Y);
  end;
305. We can lookup the value associated to the key (X,Y) leveraging the MSortedCollection. Search
function.
function IntPairKeyCollection.ObjectOf(X,Y:integer): IntPairItemPtr;
  var lPairItem: IntPairItem; I: integer;
  begin ObjectOf \leftarrow nil; lPairItem.Init(X,Y);
  if Search(addr(lpairItem), I) then ObjectOf \leftarrow Items \uparrow [I];
  end;
       This is used in justhan.pas and mizprep.pas.
function IntPairKeyCollection.FirstThat(X:integer): IntPairItemPtr;
  var I: integer;
  begin FirstThat \leftarrow nil;
  for i \leftarrow 0 to Count - 1 do
    if IntPairItemPtr(Items\uparrow[I])\uparrow.fKey.X = X then
       begin FirstThat \leftarrow Items \uparrow [I]; exit
       end;
  end;
```

Mizar Parser §307

Section 9.12. STACKED LIST OF OBJECTS

```
"Stacked" lists are really linked lists.
\langle Public interface for mobjects.pas 184 \rangle + \equiv
     { Stacked Object (List of objects) }
  StackedPtr = \uparrow StackedObj;
  StackedObj = \mathbf{object} \ (MObject)
     Previous: StackedPtr;
     constructor Init;
     \mathbf{destructor}\ \mathit{Done};\ \mathit{virtual};
  end;
308. The constructors and destructors are not implemented, so if you try to use them, just raise an error.
\langle Stacked object implementation 308\rangle \equiv
     { Stacked Object (List of objects) }
{\bf constructor}\ \mathit{StackedObj.Init};
  begin Abstract1;
  end;
{f destructor}\ StackedObj.Done;
  begin Abstract1;
  end;
This code is used in section 183.
```

 $\S 309$ Mizar Parser STRING LIST 97

Section 9.13. STRING LIST

98 STRING LIST Mizar Parser $\S 309$

```
309.
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
    { MStringList object }
  MDuplicates = (dupIgnore, dupAccept, dupError);
  PStringItem = \uparrow MStringItem;
  MStringItem = \mathbf{record} \ fString: \ PString;
    fObject: PObject:
    end:
  PStringItemList = \uparrow MStringItemList;
  MStringItemList = array [0...MaxListSize] of MStringItem;
  PStringList = \uparrow MStringList;
  MStringList = \mathbf{object} \ (MObject)
    fList: PStringItemList;
    fCount: integer;
    fCapacity: integer;
    fSorted: boolean;
    fDuplicate: MDuplicates;
    constructor Init(aCapacity : integer);
    constructor MoveStringList(var aAnother : MStringList);
             { - Internal methods- do not use them directly - }
    procedure StringListError(Code, Info: integer); virtual;
    procedure Grow;
    procedure QuickSort(L, R : integer);
    procedure ExchangeItems(Index1, Index2 : integer);
    procedure InsertItem(aIndex : integer;
         const \ aStr: String);
                                \{--\}
    procedure SetSorted(aValue : boolean);
    procedure Sort; virtual;
    function GetString(aIndex : integer): String; virtual;
    function GetObject(aIndex:integer): PObject; virtual;
    procedure PutString(aIndex : integer;
         const aStr: String); virtual;
    procedure PutObject(aIndex : integer; aObject : PObject); virtual;
    procedure SetCapacity(aCapacity: integer); virtual;
    destructor Done; virtual;
    function AddString(const aStr: String): integer; virtual;
    function AddObject(const aStr: String;
       aObject: PObject): integer; virtual;
    procedure AddStrings(var aStrings : MStringList); virtual;
    procedure Clear; virtual;
    procedure Delete(aIndex:integer); virtual;
    procedure Exchange (Index1, Index2: integer); virtual;
    procedure MoveObject(CurIndex, NewIndex : integer); virtual;
    function Find (
       const aStr: String;
       var aIndex: integer ): boolean; virtual;
    function IndexOf (const aStr: String): integer; virtual;
    function ObjectOf(const aStr: String): PObject; virtual;
    function IndexOfObject(aObject: PObject): integer;
    procedure Insert(aIndex : integer;
         const aStr: String); virtual;
```

procedure InsertObject(aIndex: integer;

§309 Mizar Parser STRING LIST 99

```
\begin{array}{c} \textbf{const} \ \ aStr \colon \ String; \\ aObject \colon \ PObject); \\ \textbf{end} \ \ ; \end{array}
```

310. Constructors. We can construct an empty string collection using *Init*. We can also move the contents of *aAnother* string collection into the caller using *MoveStringList*.

```
 \left\{ \begin{array}{ll} \text{String list implementation } 310 \right\} \equiv \\ \left\{ \begin{array}{ll} & \left\{ \begin{array}{ll} \text{MStringList} \end{array} \right. \\ \text{Constructor } \textit{MStringList.Init}(\textit{aCapacity}: integer); \\ \text{begin } \textit{MObject.Init}; \textit{fList} \leftarrow \textbf{nil}; \textit{fCount} \leftarrow 0; \textit{fCapacity} \leftarrow 0; \textit{fSorted} \leftarrow \textit{false}; \\ \textit{fDuplicate} \leftarrow \textit{dupError}; \textit{SetCapacity}(\textit{aCapacity}); \\ \text{end}; \\ \text{constructor } \textit{MStringList.MoveStringList}(\textbf{var } \textit{aAnother}: \textit{MStringList}); \\ \text{begin } \textit{MObject.Init}; \textit{fCount} \leftarrow \textit{aAnother.fCount}; \textit{fCapacity} \leftarrow \textit{aAnother.fCapacity}; \\ \textit{fSorted} \leftarrow \textit{aAnother.fSorted}; \textit{fList} \leftarrow \textit{aAnother.fList}; \textit{fDuplicate} \leftarrow \textit{aAnother.fDuplicate}; \\ \left\{ \text{Empty out the other list} \right\} \\ \textit{aAnother.fCount} \leftarrow 0; \textit{aAnother.fCapacity} \leftarrow 0; \textit{aAnother.fList} \leftarrow \textbf{nil}; \\ \text{end}; \\ \text{See also section } 315. \\ \text{This code is used in section } 183. \\ \end{array} \right.
```

311. Destructor. Since a *MStringItem* is a pointer to a string and a pointer to an *MObject*, freeing an *MStringItem* should free both of these (when they are present).

```
destructor MStringList.Done;

var I: integer;

begin inherited\ Done;

for I \leftarrow 0 to fCount - 1 do

with fList \uparrow [I] do { free fList \uparrow [I] }

begin DisposeStr(fString);

if fObject \neq \mathbf{nil} then Dispose(fObject,Done);

end;

fCount \leftarrow 0; SetCapacity(0);

end;
```

312. Adding a string. This boils down to determining the position where we will insert the new string, then inserting the string into that location, and finally returning the index to the user.

```
function MStringList.AddString (const aStr: string): integer; var lResult: integer; begin \langle Set \ lResult to the index of the newly inserted string 313\rangle; InsertItem(lResult, aStr); AddString \leftarrow lResult; end;
```

100 STRING LIST Mizar Parser §313

313. Determining the index for the string boils down to whether the collection is sorted or not. If it is unsorted, then just append the string at the end of the collection.

For sorted collections, find the location for the string. We need to give particular care when adding the new string would create a duplicate entry in the string list.

```
\langle Set lResult to the index of the newly inserted string 313\rangle \equiv if \neg fSorted then lResult \leftarrow fCount else if Find(aStr, lResult) then begin AddString \leftarrow lResult; \langle De-duplicate a string list 314\rangle; end
```

314. When we ignore duplicates (i.e., the *fDuplicate* flag is equal to *dupIgnore*), we can just terminate adding a string to the collection here.

But when we want to flag an error upon inserting a duplicate entry, then we should raise an error.

All other situations "fall through".

```
\langle De-duplicate a string list 314 \rangle \equiv case fDuplicate of dupIgnore: Exit; dupError: StringListError(coDuplicate, 0); endcases
```

This code is used in section 313.

This code is used in section 312.

315. Inserting an object. We can treat a string list as a dictionary whose keys are strings. This is because the entries are string-(pointer to object) pairs.

```
\langle String list implementation 310\rangle +\equiv function MStringList.AddObject(\mathbf{const}\ aStr:\ string;\ aObject:\ PObject):\ integer; var lResult:\ integer; begin lResult \leftarrow AddString(aStr); { Insert key } PutObject(lResult,\ aObject); { Insert value } AddObject \leftarrow lResult; { Return index } end:
```

316. Appending a string list. We can add all the entries from another *MStringList* to the caller, which is what we do in the *AddStrings* function. It does not mutate *aStrings*.

```
procedure MStringList.AddStrings(var aStrings: MStringList);
var I, r: integer;
begin for I \leftarrow 0 to aStrings.fCount - 1 do
r \leftarrow AddObject(aStrings.fList\uparrow[I].fString\uparrow, aStrings.fList\uparrow[I].fObject);
end;
```

317. Clear a string list. We can delete all the strings from a string list. This *will not* free the "values" in each key-value pair.

```
procedure MStringList.Clear;

var I: integer;

begin if fCount \neq 0 then

begin for I \leftarrow 0 to fCount - 1 do DisposeStr(fList \uparrow [I].fString);

fCount \leftarrow 0; SetCapacity(0);

end;

end;
```

 $\S318$ Mizar Parser STRING LIST 101

318. Deleting an entry by index. When given an index which is within the bounds of the caller, we free the string located at that index, decrement the size, and then shift all entries after it down by one.

```
procedure MStringList.Delete(aIndex:integer);

begin if (aIndex < 0) \lor (aIndex \ge fCount) then StringListError(coIndexError, aIndex);

DisposeStr(fList \uparrow [aIndex].fString); Dec(fCount);

if aIndex < fCount then

Move(fList \uparrow [aIndex + 1], fList \uparrow [aIndex], (fCount - aIndex) * SizeOf(MStringItem));

end;
```

319. Exchanging items. We have *Exchange* check if the indices are within the bounds of the string list, then *ExchangeItems* swaps the items around.

```
procedure MStringList.Exchange(Index1, Index2: integer);

begin if (Index1 < 0) \lor (Index1 \ge fCount) then StringListError(coIndexError, Index1);

if (Index2 < 0) \lor (Index2 \ge fCount) then StringListError(coIndexError, Index2);

ExchangeItems(Index1, Index2);

end;

procedure MStringList.ExchangeItems(Index1, Index2: integer);

var Temp: MStringItem;

begin Temp \leftarrow fList \uparrow [Index1]; fList \uparrow [Index1] \leftarrow fList \uparrow [Index2]; fList \uparrow [Index2] \leftarrow Temp;

end;
```

320. Find an entry by bisection search. We can use bisection search to find the needle in the haystack.

```
function MStringList.Find ( const aStr: string; var aIndex: integer ): boolean; var L, H, I, C: integer; lResult: boolean; begin lResult \leftarrow False; L \leftarrow 0; H \leftarrow fCount - 1; while L \leq H do

begin I \leftarrow (L+H) shr 1; C \leftarrow CompareStr(fList \uparrow [I].fString \uparrow, aStr); if C < 0 then L \leftarrow I + 1 else begin H \leftarrow I - 1; if C = 0 then

begin lResult \leftarrow True; if fDuplicate \neq dupAccept then L \leftarrow I; end; end; end; end; end; end
```

321. Reporting errors. We can propagate errors, adjusting the error code as needed. The comment here is in Polish "poprawic bledy" (which Google translates to "correct the errors")

```
procedure MStringList.StringListError(Code, Info: integer);
begin RunError(212 - Code); {! poprawic bledy}
end;
```

end;

322. Getting the string at an index. When given an index within bounds, we try to get the string located there. If there is no string located at that entry, return the empty string.

102 STRING LIST Mizar Parser $\S 323$

323. Get object at index. We can get the object at an index, provided it is within bounds.

```
function MStringList.GetObject(aIndex:integer): PObject;

begin if (aIndex < 0) \lor (aIndex \ge fCount) then StringListError(coIndexError, aIndex);

GetObject \leftarrow fList \uparrow [aIndex].fObject;

end:
```

324. Ensure capacity for string lists. The growth rate for string lists differs from the earlier discussion on the growth rate for dynamic arrays. Well, actually, recalling our discussion ($\S196$), we find this is identical to the previous growth rate. So I am not sure why this code is duplicated.

```
procedure MStringList.Grow;
var Delta: integer;
begin if fCapacity > 64 then Delta \leftarrow fCapacity div 4
else if fCapacity > 8 then Delta \leftarrow 16
else Delta \leftarrow 4;
SetCapacity(fCapacity + Delta);
end;
```

325. Index of a string. There are two branches to this function: one for unsorted string lists, the second for sorted string lists.

```
function MStringList.IndexOf (const aStr: string): integer; var lResult: integer; begin if \neg fSorted then

begin for lResult \leftarrow 0 to fCount - 1 do

if CompareStr(fList\uparrow[lResult].fString\uparrow, aStr) = 0 then

begin IndexOf \leftarrow lResult; Exit; end;

lResult \leftarrow -1;
end

else if \neg Find(aStr, lResult) then lResult \leftarrow -1;
{ Assert: lResult = -1 if aStr is not in the caller }

IndexOf \leftarrow lResult;
end;
```

326. Value for a key. This appears to duplicate code from GetObject (§323).

```
function MStringList.ObjectOf (const aStr: string): PObject; var I: integer; begin ObjectOf \leftarrow nil; I \leftarrow IndexOf(aStr); if I \geq 0 then ObjectOf \leftarrow fList \uparrow [I].fObject; end;
```

327. Insert a string at an index. This seems to involve duplicate code as *AddString* (§312), but allows duplicate entries (which might violate the invariants of a string list).

```
procedure MStringList.Insert(aIndex:integer;

const aStr:string);

begin if fSorted then StringListError(coSortedListError, 0);

if (aIndex < 0) \lor (aIndex > fCount) then StringListError(coIndexError, aIndex);

InsertItem(aIndex, aStr);

end;
```

§328 Mizar Parser STRING LIST 103

328. Inserting an item at an index. We ensure the capacity of the string list. Then we shift the entries to the right by 1, if needed. We insert the string associated with no object. Then increment the logical size of the dynamic array.

```
procedure MStringList.InsertItem(aIndex:integer; const aStr:string);
begin if fCount = fCapacity then Grow;
{ Shift existing entries to right by 1 }
if aIndex < fCount then
Move(fList\uparrow[aIndex], fList\uparrow[aIndex+1], (fCount - aIndex) * SizeOf(MStringItem));
with fList\uparrow[aIndex] do
begin fObject \leftarrow nil; fString \leftarrow NewStr(aStr); end;
inc(fCount);
end;
```

329. Find the index for an object. Find the first instance of a key-value entry whose value is equal to the given object. If the given object is absent from the string list, return -1.

```
function MStringList.IndexOfObject(aObject: PObject): integer; var lResult: integer; begin for lResult \leftarrow 0 to fCount - 1 do

if GetObject(lResult) = aObject then

begin IndexOfObject \leftarrow lResult; Exit; end;
IndexOfObject \leftarrow -1; end;
```

330. Inserting an object associated with a string.

```
procedure MStringList.InsertObject(aIndex : integer;
    const aStr: string;
    aObject: PObject);
begin Insert(aIndex, aStr); PutObject(aIndex, aObject);
end;
```

331. Moving a key-value entry around. We can take the key-value entry at *CurIndex*, remove it from the string list, then insert it at *NewIndex*. It is important to note: the *NewIndex* is the index *after* the delete operation has occurred.

```
procedure MStringList.MoveObject(CurIndex, NewIndex : integer);
var TempObject: PObject; TempString: string;
begin if CurIndex \neq NewIndex then
begin TempString \leftarrow GetString(CurIndex); TempObject \leftarrow GetObject(CurIndex);
Delete(CurIndex); InsertObject(NewIndex, TempString, TempObject);
end;
end;
```

104 STRING LIST Mizar Parser $\S 332$

332. Inserting a string at an index. Well, if this is a sorted collection, then raise an error: you can't insert strings willy-nilly!

Then check the index is within bounds, raise an error for out-of-bounds indices.

Then mutate the entry at aIndex to have its string be equal to NewStr(aStr).

This will always mutate the caller, even when the string located at the entry indexed by aIndex is identical to aStr.

```
procedure MStringList.PutString(aIndex:integer; const aStr:string); begin if fSorted then StringListError(coSortedListError, 0); if (aIndex < 0) \lor (aIndex \ge fCount) then StringListError(coIndexError, aIndex); fList \uparrow [aIndex].fString \leftarrow NewStr(aStr); end:
```

333. Inserting an object at an index. When given an index within bounds of the caller's underlying array, mutate its object to be the given *aObject*. Again, this *always* mutates the caller.

```
procedure MStringList.PutObject(aIndex:integer; aObject:PObject); begin if (aIndex < 0) \lor (aIndex \ge fCount) then StringListError(coIndexError, aIndex); fList \uparrow [aIndex].fObject \leftarrow aObject; end;
```

334. Quicksorting a string list. See also §257 and §403.

```
procedure MStringList.QuickSort(L,R:integer);

var I,J:integer;\ P:string;

begin repeat I\leftarrow L;\ J\leftarrow R;\ P\leftarrow fList\uparrow[(L+R)\ shr\ 1].fString\uparrow;

repeat while CompareStr(fList\uparrow[I].fString\uparrow,P)<0 do inc(I);

while CompareStr(fList\uparrow[J].fString\uparrow,P)>0 do Dec(J);

if I\leq J then

begin ExchangeItems(I,J);\ inc(I);\ Dec(J); end;

until I>J;

if L< J then QuickSort(L,J);

L\leftarrow I;

until I\geq R;

end;
```

335. Changing the capacity of a string list. Of particular note here, changing the capacity of a string list *does not* delete anything. That work must be delegated elsewhere when aCapacity < Self.fCapacity (if that case ever occurs).

```
procedure MStringList.SetCapacity(aCapacity:integer);
var lList: PStringItemList;
begin if aCapacity < fCount then aCapacity \leftarrow fCount;
if aCapacity > MaxListSize then aCapacity \leftarrow MaxListSize;
if aCapacity \neq fCapacity then
begin if aCapacity = 0 then lList \leftarrow nil
else begin GetMem(lList, aCapacity * SizeOf(MStringItem));
if (fCount \neq 0) \land (fList \neq nil) then Move(fList \uparrow, lList \uparrow, fCount * SizeOf(MStringItem));
end;
if fCapacity \neq 0 then FreeMem(fList, fCapacity * SizeOf(MStringItem));
fList \leftarrow lList; fCapacity \leftarrow aCapacity;
end; {ReallocMem(fList, NewCapacity * SizeOf(MStringItem)); fCapacity := NewCapacity;}
end;
```

 $\S 336$ Mizar Parser STRING LIST 105

336. Toggle 'sorted' flag. Allow the user to toggle the "sorted" flag. When toggled to *True*, be sure to sort the string list.

```
procedure MStringList.SetSorted(aValue:boolean);
begin if fSorted \neq aValue then
begin if aValue then Sort;
fSorted \leftarrow aValue;
end;
end;
```

337. Sorting. This is a wrapper around the quicksort function ($\S 334$), invoked when the *fSorted* flag is false.

This appears to be used in the SetSorted procedure, but that is not used anywhere.

```
procedure MStringList.Sort;
begin if \neg fSorted \land (fCount > 1) then
begin fSorted \leftarrow true; QuickSort(0, fCount - 1);
end;
end;
```

338. Allocating a new string. Allocating a new PString from a string. When the empty string is given, return nil. Otherwise allocate a new block of memory in the Heap, then set its contents equal to S.

```
{ Dynamic string handling routines } function NewStr(\mathbf{const}\ S:\ String):\ PString; var P:\ PString; begin if S=\ \ then\ P\leftarrow nil else begin GetMem(P,length(S)+1);\ P\uparrow\leftarrow S; end; NewStr\leftarrow P; end;
```

339. Deleting a string. A convenience function to avoid accidentally freeing a nil string pointer.

```
procedure DisposeStr(P:PString);
begin if P \neq nil then FreeMem(P, length(P\uparrow) + 1);
end;
```

106 TUPLES OF INTEGERS Mizar Parser $\S340$

Section 9.14. TUPLES OF INTEGERS

```
340.
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
    { Partial integers Functions }
  IntTriplet = \mathbf{record} \ X1, X2, Y: integer;
    end:
const\ MaxIntPairSize = MaxSize\ div\ SizeOf\ (IntPair);
  MaxIntTripletSize = MaxSize \ div \ SizeOf(IntTriplet);
341. Now, this is the remainder of the interface
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
type IntPairListPtr = \uparrow IntPairList; IntPairList = \mathbf{array} \ [0 ... MaxIntPairSize - 1] of IntPair;
  IntPairSeqPtr = \uparrow BinIntFunc; IntPairSeq = \mathbf{object} \ (MObject)
    Items: IntPairListPtr;
    Count: integer;
    Limit: integer;
    constructor Init(aLimit : integer);
    procedure NatSetError(Code, Info: integer); virtual;
    destructor Done; virtual;
    procedure SetLimit(aLimit: integer); virtual;
    procedure Insert(const aItem: IntPair); virtual;
    procedure AtDelete(aIndex:integer);
    procedure DeleteAll;
    procedure AssignPair(X, Y : integer); virtual;
  end:
342. First, we have a helper function for flagging errors.
\langle \text{ Tuples of integers } 342 \rangle \equiv
    { Pairs of an integers }
procedure IntPairSeq.NatSetError(Code, Info: integer);
  begin RunError(212 - Code); end;
343. Constructor.
constructor IntPairSeq.Init(aLimit: integer);
  begin MObject.Init; Items \leftarrow nil; Count \leftarrow 0; Limit \leftarrow 0; SetLimit(aLimit);
  end:
344. Destructor.
destructor IntPairSeq.Done;
  begin Count \leftarrow 0; SetLimit(0);
  end;
345. Insert an element.
procedure IntPairSeq.Insert(const aItem: IntPair);
  begin if Count \geq MaxIntPairSize then NatSetError(coOverflow, 0);
  if Limit = Count then SetLimit(Limit + GrowLimit(Limit));
  Items \uparrow [Count] \leftarrow aItem; inc(Count);
  end;
```

 $\S346$ Mizar Parser TUPLES OF INTEGERS 107

Delete an element at an index. procedure IntPairSeq.AtDelete(aIndex : integer); **var** *i*: *integer*; begin if $(aIndex < 0) \lor (aIndex > Count)$ then **begin** NatSetError(coIndexError, 0); exit; end; if aIndex < Count - 1 then for $i \leftarrow aIndex$ to Count - 2 do $Items \uparrow [i] \leftarrow Items \uparrow [i+1];$ Dec(Count);end; 347. Resizing an IntPair sequence. **procedure** IntPairSeq.SetLimit(aLimit:integer); var aItems: IntPairListPtr; **begin if** aLimit < Count **then** $aLimit \leftarrow Count$; if aLimit > MaxIntPairSize then $ALimit \leftarrow MaxIntPairSize$; if $aLimit \neq Limit$ then begin if ALimit = 0 then $AItems \leftarrow nil$ else begin GetMem(AItems, ALimit * SizeOf(IntPair));if $(Count \neq 0) \land (Items \neq nil)$ then $Move(Items \uparrow, aItems \uparrow, Count * SizeOf(IntPair));$ if $Limit \neq 0$ then FreeMem(Items, Limit * SizeOf(IntPair)); $Items \leftarrow aItems; \ Limit \leftarrow aLimit;$ end; end; 348. Deleting all entries. We just set the logical size to zero. It leaves everything else untouched.

348. Deleting all entries. We just set the logical size to zero. It leaves everything else untouched procedure IntPairSeq.DeleteAll;
begin Count ← 0; end;

349. Append a pair of integers. We create a new IntPair using X and Y, then append it to the caller. procedure IntPairSeq.AssignPair(X,Y:integer); var IIntPair: IntPair; begin $IIntPair.X \leftarrow X$; $IIntPair.Y \leftarrow Y$; Insert(IIntPair);

end;

108 INT REL Mizar Parser $\S 350$

Section 9.15. INT REL

```
This is used in the iocorrel.pas, identify.pas, equalizer.pas, the analyzer, and a polynomial
library.
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
  IntRelPtr = \uparrow IntRel;
  IntRel = \mathbf{object} (IntPairSeq)
    constructor Init(aLimit : integer);
    procedure Insert(const aItem: IntPair); virtual;
    procedure AtInsert(aIndex : integer;
         const aItem: IntPair); virtual;
    function Search(X, Y : integer; var aIndex : integer): boolean; virtual;
    function IndexOf(X, Y : integer): integer;
    constructor CopyIntRel(var aFunc : IntRel);
    function IsMember(X, Y : integer): boolean; virtual;
    procedure AssignPair(X, Y : integer); virtual;
  end;
351. Constructor. This is just the inherited constructor.
\langle \text{Int relation implementation } 351 \rangle \equiv
{ IntRel }
constructor IntRel.Init(aLimit: integer);
  begin inherited Init(aLimit);
  end:
This code is used in section 183.
       Inserting an entry.
procedure IntRel.Insert(const aItem: IntPair);
  var I: integer;
  begin if \neg Search(aItem.X, aItem.Y, I) then
    begin if (I < 0) \lor (I > Count) then
       begin NatSetError(coIndexError, 0); exit; end;
    if Count \geq MaxIntPairSize then NatSetError(coOverflow, 0);
         { Finished with the possible errors }
    if Limit = Count then SetLimit(Limit + GrowLimit(Limit));
    if I \neq Count then Move(Items \uparrow [I], Items \uparrow [I+1], (Count - I) * SizeOf(IntPair));
    Items \uparrow [I] \leftarrow aItem; inc(Count);
    end;
end;
```

 $\S353$ Mizar Parser INT REL 109

```
353.
       Insert at a specific index.
procedure IntRel.AtInsert(aIndex: integer;
    const aItem: IntPair);
  begin if (aIndex < 0) \lor (aIndex > Count) then NatSetError(coIndexError, aIndex);
  if Count = Limit then SetLimit(Limit + GrowLimit(Limit));
       { Shift everything to the right by 1 }
  if aIndex < Limit then Move(Items \uparrow [aIndex], Items \uparrow [aIndex + 1], (Count - aIndex) * SizeOf(IntPair));
       { Update the items, increment the logical size }
  Items \uparrow [aIndex] \leftarrow aItem; inc(Count);
  end:
354. Bisection search for a relation. Search through IntRel for a relation X = Y. Note that this is
not symmetric, i.e., if we have Y = X in the IntRel, then it will not match.
  Mutates the aIndex. If the relation is missing, aIndex will return where it should be.
function IntRel.Search(X, Y : integer; var aIndex : integer): boolean;
  var L, H, I, C: integer;
  begin Search \leftarrow False; L \leftarrow 0; H \leftarrow Count - 1;
  while L \leq H do
    begin I \leftarrow (L+H) shr 1; C \leftarrow CompareIntPairs(Items\uparrow[I].X, Items\uparrow[I].Y, X, Y);
    if C < 0 then L \leftarrow I + 1
    else begin H \leftarrow I - 1;
       if C = 0 then
         begin Search \leftarrow True; L \leftarrow I; end;
       end;
    end:
  aIndex \leftarrow L;
  end;
355. Copy constructor. This moves the contents of aFunc into the caller. It will mutate the caller but
not the argument supplied. The Move function copies the contents of one region of memory to another.
constructor IntRel.CopyIntRel(var aFunc : IntRel);
  begin Init(aFunc.Limit); Move(aFunc.Items \uparrow, Items \uparrow, aFunc.Limit * SizeOf(IntPair));
  Count \leftarrow aFunc.Count;
  end;
356. Index of a relation. This will return the index of the X = Y entry. If it is absent from the caller,
then return -1.
function IntRel.IndexOf(X, Y : integer): integer;
  var I: integer;
  begin IndexOf \leftarrow -1;
  if Search(X, Y, I) then IndexOf \leftarrow I;
  end:
       Test for membership. This just tests if X = Y is contained in the caller.
357.
function IntRel.IsMember(X, Y : integer): boolean;
  var I: integer;
  begin IsMember \leftarrow Search(X, Y, I); end;
```

110 INT REL Mizar Parser $\S 358$

```
358. 
 procedure IntRel.AssignPair(X,Y:integer); begin if IsMember(X,Y) then exit; inheritedAssignPair(X,Y); end;
```

Section 9.16. PARTIAL INTEGERS FUNCTIONS

```
359.
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
  NatSetPtr = \uparrow NatSet;
  NatSet = \mathbf{object} \ (IntRel)
    Delta: integer;
    Duplicates: boolean;
    constructor Init(aLimit, aDelta : integer);
    constructor InitWithElement(X:integer);
    destructor Done; virtual;
    procedure Insert(const aItem: IntPair); virtual;
    function SearchPair(X : integer; var Index : integer): boolean; virtual;
    function ElemNr(X:integer): integer;
    constructor CopyNatSet(const fFunc: NatSet);
    procedure InsertElem(X : integer); virtual;
    procedure DeleteElem(fElem : integer); virtual;
    procedure EnlargeBy(const fAnother: NatSet); {? virtual;?}
    procedure ComplementOf (const fAnother: NatSet);
    procedure Intersect With (const fAnother: NatSet);
    function HasInDom(fElem:integer): boolean; virtual;
    function IsEqualTo(const fFunc: NatSet): boolean;
    function IsSubsetOf (const fFunc: NatSet): boolean;
    function IsSupersetOf (const fFunc: NatSet): boolean;
    function Misses(const fFunc: NatSet): boolean;
    constructor MoveNatSet(var fFunc : NatSet);
  end:
360.
       Constructor. The empty NatSet can be constructed with the usual initialization.
\langle \text{ Partial integer function implementation } 360 \rangle \equiv
    { Partial integers Functions }
constructor NatSet.Init(aLimit, aDelta: integer);
  begin MObject.Init; Items \leftarrow \mathbf{nil}; Count \leftarrow 0; Limit \leftarrow 0; Delta \leftarrow ADelta; SetLimit(ALimit);
  Duplicates \leftarrow False;
  end:
This code is used in section 183.
361. Singleton constructor. This initializes the Delta set to 4, and the aLimit set to 0. Then insert
the given integer.
constructor NatSet.InitWithElement(X:integer);
  begin Init(0,4); InsertElem(X);
  end;
       Destructor. This delegates the heavy work to SetLimit(0).
destructor NatSet.Done;
  begin Count \leftarrow 0; SetLimit(0);
  end:
```

end;

363. Inserting a pair of integers. Using Search to find where to insert X = Y, possibly growing the underlying array if needed.

```
procedure NatSet.Insert(const aItem: IntPair);
  var I: integer;
  begin if \neg SearchPair(aItem.X, I) \lor Duplicates then
    begin if (I < 0) \lor (I > Count) then { Out of bounds, raise an error }
       begin NatSetError(coIndexError, 0); exit; end;
    if Limit = Count then { Grow the capacity, if possible }
       begin if Delta = 0 then
         begin NatSetError(coOverFlow, 0); exit; end;
       SetLimit(Limit + Delta);
       end:
    if I \neq Count then Move(Items \uparrow [I], Items \uparrow [I+1], (Count-I) * SizeOf(IntPair));
    Items \uparrow [I] \leftarrow aItem; inc(Count);
    end;
  end:
       Equality of IntPair objects. This just tests the componentwise equality of two IntPair objects.
function Equals (Key1, Key2 : IntPair): boolean;
  begin Equals \leftarrow (Key1.X = Key2.X) \wedge (Key1.Y = Key2.Y);
  end;
       Search. This is a bisection search for any relation of the form X = Y for some Y.
365.
function NatSet.SearchPair(X:integer; var Index:integer): boolean;
  var L, H, I, C: integer;
  begin SearchPair \leftarrow False; L \leftarrow 0; H \leftarrow Count - 1;
  while L \leq H do
    begin I \leftarrow (L+H) shr 1; C \leftarrow CompareInt(Items \uparrow [I].X, X);
    if C < 0 then L \leftarrow I + 1
    else begin H \leftarrow I - 1;
       if C = 0 then
         begin SearchPair \leftarrow True;
         if \neg Duplicates then L \leftarrow I;
         end;
       end;
    end;
  Index \leftarrow L;
  end:
       Copy constructor. We can copy the contents of another NatSet into the caller. This mutates the
caller, but leaves the given NatSet unchanged.
constructor NatSet.CopyNatSet(const fFunc: NatSet);
  begin Init(fFunc.Limit, fFunc.Delta); Move(fFunc.Items \uparrow, Items \uparrow, fFunc.Limit * SizeOf(IntPair));
  Count \leftarrow fFunc.Count;
```

var I: integer;

end;

begin $HasInDom \leftarrow SearchPair(fElem, I);$

Move constructor. We can also move the contents of another NatSet into the caller. This will mutate the other *NatSet* to have **nil** items and 0 capacity. constructor NatSet.MoveNatSet(var fFunc : NatSet); **begin** $Init(fFunc.Limit, fFunc.Delta); Self \leftarrow fFunc; fFunc.DeleteAll; fFunc.Limit <math>\leftarrow 0;$ $fFunc.Items \leftarrow \mathbf{nil};$ end; **368.** Union operation. We can merge another *NatSet* into the caller. **procedure** NatSet.EnlargeBy(const fAnother: NatSet); **var** *I*: *integer*; **begin for** $I \leftarrow 0$ **to** fAnother.Count - 1 **do** $InsertElem(fAnother.Items \uparrow [i].X);$ end: **Set complement.** We can destructively remove from the caller all elements appearing in fAnother nat set. **procedure** NatSet.ComplementOf(**const** fAnother: NatSet); var I: integer; **begin for** $I \leftarrow 0$ **to** fAnother.Count - 1 **do** $DeleteElem(fAnother.Items \uparrow [i].X);$ end; **Take intersection.** This computes $Self \leftarrow Self \cap Other$ procedure NatSet.IntersectWith(const fAnother: NatSet); **var** k: integer; **begin** $k \leftarrow 0$; while k < Count doif $\neg fAnother.HasInDom(Items\uparrow[k].X)$ then AtDelete(k)else inc(k); end; **371.** Insert an element. We can insert X=0 into the caller. procedure NatSet.InsertElem(X : integer); var lIntPair: IntPair: **begin** $UntPair.X \leftarrow X$; $UntPair.Y \leftarrow 0$; Insert(UntPair); end: **372.** Deleting an element. Similarly, we can delete the first element of the form X = Y for some Y. **procedure** NatSet.DeleteElem(fElem: integer); **var** *I*: *integer*; **begin if** SearchPair(fElem, I) **then** AtDelete(I); end: **373.** We can test if an element X is in the domain of the caller. **function** NatSet.HasInDom(fElem: integer): boolean;

114

374. Set equality predicate. This assumes that there are no duplicate entries in a NatSet data structure.

```
function NatSet.IsEqualTo(\mathbf{const}\ fFunc:\ NatSet): boolean;
var I:\ integer;
begin IsEqualTo \leftarrow false;
if Count \neq fFunc.Count\ \mathbf{then}\ exit;
for I \leftarrow 0\ \mathbf{to}\ Count - 1\ \mathbf{do}
if \neg Equals(Items \uparrow [I], fFunc.Items \uparrow [I])\ \mathbf{then}\ exit;
IsEqualTo \leftarrow true;
end;
```

375. Subset predicate. The comment is Polish for (according to Google translate): "If we're checking if a small function is contained within a large one, commenting it out might be better." There is a commented out function which I removed.

```
function NatSet.IsSubsetOf (const fFunc: NatSet): boolean;
var i, j, k, c: integer; { Jezeli sprawdzamy, czy mala funkcja jest zawarta w duzej, to to wykomentowane moze byc lepsze }

begin IsSubsetOf \leftarrow false; c \leftarrow fFunc.Count;
if c < Count then exit;
j \leftarrow 0;
for i \leftarrow 0 to Count - 1 do
begin k \leftarrow Items\uparrow[i].X;
while (j < c) \land (fFunc.Items\uparrow[j].X < k) do inc(j);
if (j = c) \lor \neg Equals(fFunc.Items\uparrow[j], Items\uparrow[i]) then exit;
end;
IsSubsetOf \leftarrow true;
end;
```

376. Superset predicate. This just takes advantage of the fact that $Y \supseteq X$ is the same as $X \subseteq Y$, then use the subset predicate.

```
function NatSet.IsSupersetOf (const fFunc: NatSet): boolean;
begin IsSupersetOf \leftarrow fFunc.IsSubsetOf (Self);
end;
```

377. Test if two sets are disjoint. This iterates over the smaller of the two sets, checking if every element in the smaller set *does not* appear in the larger set.

```
function NatSet.Misses (const fFunc: NatSet): boolean;

var I,k: integer;

begin if Count > fFunc.Count then

begin for k \leftarrow 0 to fFunc.Count - 1 do

if SearchPair(fFunc.Items \uparrow [k].X, I) then

begin Misses \leftarrow false; exit end

end

else begin for k \leftarrow 0 to Count - 1 do

if fFunc.SearchPair(Items \uparrow [k].X, I) then

begin Misses \leftarrow false; exit end;

end;

Misses \leftarrow true;

end;
```

378. Index for an element. This searches for the index associated with relations of the form X = Y. If any such relation appears, return its index. Otherwise, return -1.

It leaves the caller unmodified, so it is a pure function.

```
 \begin{array}{ll} \textbf{function} & NatSet.ElemNr(X:integer): integer;\\ \textbf{var} & I: integer;\\ \textbf{begin} & ElemNr \leftarrow -1;\\ \textbf{if} & SearchPair(X,I) \textbf{ then} & ElemNr \leftarrow I;\\ \textbf{end}; \end{array}
```

Section 9.17. FUNCTION OF NATURAL NUMBERS

```
The NatFunc is used in the analyzer, equalizer, unifier, and elsewhere. Its destructor is the only
place where nConsistent \leftarrow false.
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
  NatFuncPtr = \uparrow NatFunc;
  NatFunc = \mathbf{object} \ (NatSet)
    nConsistent: boolean:
    constructor InitNatFunc(ALimit, ADelta: integer);
    constructor CopyNatFunc(const fFunc: NatFunc);
    constructor MoveNatFunc(var fFunc : NatFunc);
    constructor LCM (const aFunc1, aFunc2: NatFunc);
    procedure Assign(X, Y : integer); virtual;
    procedure Up(X:integer); virtual;
    procedure Down(X:integer); virtual;
    function Value (fElem: integer): integer; virtual;
    procedure Join(const fFunc: NatFunc);
    destructor Refuted; virtual;
    procedure EnlargeBy(fAnother : NatFuncPtr); {? virtual;?}
    function JoinAtom(fLatAtom: NatFuncPtr): NatFuncPtr;
    function Compare With (const fNatFunc: NatFunc): integer;
    function WeakerThan(const fNatFunc: NatFunc): boolean;
    function IsMultipleOf (const fNatFunc: NatFunc): boolean;
    procedure Add(const aFunc: NatFunc);
    function CountAll: integer; virtual;
  end;
      Constructors. We have the basic constructors for an empty NatFunc, a copy constructor, and a
move constructor. The move constructor is destructive on the supplied argument.
\langle NatFunc \text{ implementation } 380 \rangle \equiv
constructor NatFunc.InitNatFunc(ALimit, ADelta: integer);
  begin inherited Init(ALimit, ADelta); nConsistent \leftarrow true;
  end;
constructor NatFunc.CopyNatFunc(const fFunc: NatFunc);
  begin Init(fFunc.Limit, fFunc.Delta); Move(fFunc.Items \uparrow, Items \uparrow, fFunc.Limit * SizeOf(IntPair));
  Count \leftarrow \textit{fFunc.Count}; \ \textit{nConsistent} \leftarrow \textit{fFunc.nConsistent};
  end:
constructor NatFunc.MoveNatFunc(var fFunc : NatFunc);
  begin Init(fFunc.Limit, fFunc.Delta); Self \leftarrow fFunc; fFunc.DeleteAll; fFunc.Limit \leftarrow 0;
  fFunc.Items \leftarrow \mathbf{nil};
  end:
See also section 396.
This code is used in section 183.
```

end;

Mizar Parser

381. Constructor (LCM). The least common multiple between two NatFunc objects is another way to construct a NatFunc instance. This seems to be the LCM in the sense of commutative rings (if x and y are elements of a commutative ring R, then lcm(x,y) is such that x divides lcm(x,y) and y divides lcm(x,y) — moreover, lcm(x,y) is the smallest such quantity, in the sense that lcm(x,y) divides any other such quantity).

 $lcm(f, g) = \{ (x, y) \mid \exists y_1, y_2, (x, y_1) \in f, (x, y_2) \in g, y = lcm(y_1, y_2) \},$

```
For the ring (or ringoid) N^N, this amounts to
```

```
with the condition that when y_1 = 0, y = y_2 (and similarly y_2 = 0 implies y = y_1).
constructor NatFunc.LCM (const aFunc1, aFunc2: NatFunc);
  var i, j, m: integer;
  begin m \leftarrow aFunc2.Delta;
  if aFunc1.Delta > m then m \leftarrow aFunc1.Delta;
  InitNatFunc(aFunc1.Limit + aFunc2.Limit, m); i \leftarrow 0; j \leftarrow 0;
  while (i < aFunc1.Count) \land (j < aFunc2.Count) do
     case CompareInt(aFunc1.Items\uparrow[i].X, aFunc2.Items\uparrow[j].X) of
     -1: begin Insert(aFunc1.Items \uparrow [i]); inc(i) end;
     0: begin
                 \{ m = \max(f(i), g(i)) \}
       m \leftarrow aFunc1.Items \uparrow [i].Y;
       if aFunc2.Items\uparrow[j].Y > m then m \leftarrow aFunc2.Items\uparrow[j].Y;
       Assign(aFunc1.Items \uparrow [i].X, m); \{ destructively set f(i) \leftarrow m \}
       inc(i); inc(j);
       end;
     1: begin Insert(aFunc2.Items \uparrow [j]); inc(j) end;
     endcases;
  if i \geq aFunc1.Count then
     for j \leftarrow j to aFunc2.Count - 1 do Insert(aFunc2.Items \uparrow [j])
  else for i \leftarrow i to aFunc1.Count - 1 do Insert(aFunc1.Items \uparrow [i]);
```

382. Extend a natural function. We can extend a natural function to assign a value y to a place where it is not yet defined $x \notin \text{dom}(f)$.

We should recall HasInDom (§373) which depends on SearchPair (§365) is relevant. When trying to assign a different value y to an already defined $f(x) \neq y$, then we have refuted something.

```
procedure NatFunc.Assign(X,Y:integer);
var lIntPair: IntPair;
begin if nConsistent then
begin if HasInDom(X) \land (Value(X) \neq Y) then
begin Refuted; \ exit \ \mathbf{end};
lIntPair.X \leftarrow X; \ lIntPair.Y \leftarrow Y; \ Insert(lIntPair);
end;
end;
```

begin inherited Done; $nConsistent \leftarrow false$

end;

```
Increment f(x). Given a NatFunc object f, and an integer x, f.Up(x) will
(1) If x \in \text{dom}(f), then update the value f(x) \ge f(x) + 1
(2) Otherwise, x \notin \text{dom}(f), so this corresponds to f(x) = 0, then we mutate f(x) \leftarrow 1.
procedure NatFunc.Up(X:integer);
  var I: integer; lIntPair: IntPair;
  begin if nConsistent then
    begin if SearchPair(X, I) then inc(Items \uparrow [I].Y)
    else begin lIntPair.X \leftarrow X; lIntPair.Y \leftarrow 1; Insert(lIntPair);
    end:
  end;
384. Decrement f(x). Given a NatFunc object f, and an integer x, f.Down(x) will
(1) If x \in \text{dom}(f), then update the value f(x) \geq f(x) - 1 and if this is then zero, remove it from the
    function.
(2) Otherwise, x \notin \text{dom}(f), so this corresponds to f(x) = 0, and we cannot mutate f(x) \leftarrow -1 without
    making it no longer natural-valued. So we raise an error.
procedure NatFunc.Down(X:integer);
  var I: integer;
  begin if nConsistent then
    begin if SearchPair(X, I) then
       begin dec(Items \uparrow [I].Y);
       if Items \uparrow [I].Y = 0 then AtDelete(I);
    else NatSetError(coConsistentError, 0);
    end;
  end;
385. Getting the value of f(x) when x \in \text{dom}(f). When x \notin \text{dom}(f), raise an error.
function NatFunc. Value (fElem: integer): integer;
  var I: integer;
  begin if SearchPair(fElem, I) then Value \leftarrow Items\uparrow[I].Y
  else NatSetError(coDuplicate, 0);
  end;
386. Destructor. We usually try to extend partial functions on N, but if we end up trying to extend
where it is already defined to a different value, then we arrive at an inconsistent extension. It is referred to
as a "refuted" situation.
destructor NatFunc.Refuted;
```

387. Join. For two partial functions $f: \mathbb{N} \to \mathbb{N}$ and $g: \mathbb{N} \to \mathbb{N}$, we form $f \cup g$ provided

```
f \cap g = f|_{\text{dom}(f \cap g)} = g|_{\text{dom}(f \cap g)}.
```

That is to say, for all $x \in \text{dom}(f) \cap \text{dom}(g)$, we have f(x) = g(x).

The comment is in Polish, which Google translates as: "It seems that the *Join* and *EnlargeBy* procedures below do the same thing. *EnlargeBy* should be faster for small collections. If not, it's not worth the code waste and can be discarded. On the other hand, these procedures are primarily intended for (very) small collections."

Also worth observing, this tests for consistency in the other *NatFunc*.

```
{ Wyglada na to, ze ponizej podane procedury "Join" i "EnlargeBy" robia to samo, "EnlargeBy"
      powinna byc szybsza dla malych kolekcji. Jezeli tak nie jest nie warto tracic kodu i mozna ja
       wyrzucic. Z drugiej strony procedury te maja byc glownie stosowane do (bardzo) malych kolekcji.
procedure NatFunc.Join(const fFunc: NatFunc);
  var I, k: integer;
  begin if nConsistent then
    begin if \neg fFunc.nConsistent then
       begin Refuted; exit end;
    for k \leftarrow 0 to fFunc.Count - 1 do
       if SearchPair(fFunc.Items \uparrow [k].X, I) then
         begin if \neg Equals(Items \uparrow [I], fFunc.Items \uparrow [k]) then
           begin Refuted; exit end:
         end
       else Insert(fFunc.Items \uparrow [k]);
    end;
  end;
```

388. This function performs the same task as the previous one (i.e., it merges another partial function into the caller, provided it is consistent on overlap).

```
procedure NatFunc.EnlargeBy(fAnother : NatFuncPtr); {? virtual;?}
  var i, j, lCount, lLimit: integer; lItems: IntPairListPtr;
  begin if nConsistent then
     begin if \neg fAnother \uparrow . nConsistent then
        begin Refuted; exit end;
     if fAnother \uparrow. Count = 0 then exit;
     lCount \leftarrow Count; \ lItems \leftarrow Items; \ lLimit \leftarrow Limit; \ Limit \leftarrow 0; \ Count \leftarrow 0;
     SetLimit(lCount + fAnother \uparrow. Count); i \leftarrow 0; j \leftarrow 0;
     while (i < lCount) \land (j < fAnother \uparrow. Count) do
        case CompareInt(lItems\uparrow[i].X, fAnother\uparrow.Items\uparrow[j].X) of
        -1: begin Insert(lItems \uparrow [i]); inc(i) end;
        0: begin if Equals(lItems\uparrow[i], fAnother\uparrow.Items\uparrow[j]) then Insert(lItems\uparrow[i])
          else begin Refuted; FreeMem(lItems, lLimit * SizeOf(IntPair)); exit end;
           inc(i); inc(j);
          end;
        1: begin Insert(fAnother \uparrow . Items \uparrow [j]); inc(j) end;
        endcases;
     if i \geq lCount then
        for j \leftarrow j to fAnother \uparrow. Count - 1 do Insert(fAnother \uparrow. Items \uparrow [j])
     else for i \leftarrow i to lCount - 1 do Insert(lItems \uparrow [i]);
     SetLimit(0); FreeMem(lItems, lLimit * SizeOf(IntPair));
     end;
  end;
```

end;

389. We want to join two partial functions $f: \mathbb{N} \to \mathbb{N}$ and $g: \mathbb{N} \to \mathbb{N}$ without accidentally mutating either f or g to be refuted. To do this, we copy the caller, then enlarge it with the other partial function. If the result is consistent, then return it. Otherwise, return **nil**.

This leaves both the caller and *fLatAtom* unchanged, so it's a pure function.

```
function NatFunc.JoinAtom(fLatAtom: NatFuncPtr): NatFuncPtr;
var lEval: NatFunc;
begin JoinAtom \leftarrow nil; lEval.CopyNatFunc(Self); lEval.EnlargeBy(fLatAtom);
if lEval.nConsistent then JoinAtom \leftarrow NatFuncPtr(lEval.CopyObject);
end;
```

390. Comparing partial functions. Given two partial functions, $f: \mathbb{N} \to \mathbb{N}$ and $g: \mathbb{N} \to \mathbb{N}$, we want to compare them. We first start with comparing ||f|| against ||g||. If they are not equal, then this is the result. When ||f|| = ||g||, iterate through each $x \in \text{dom}(f)$, and then compare f(x) against g(x). If f(x) < g(x), then return -1. If f(x) > g(x), then return +1. Otherwise keep iterating until we have examined all of dom(f), and then we return 0.

```
function CompareNatFunc(aKey1, aKey2 : Pointer): integer;
  var i, lInt: integer;
  begin with NatFuncPtr(aKey1)\uparrow do
     begin lInt \leftarrow CompareInt(Count, NatFuncPtr(aKey2)\uparrow.Count);
     if lInt \neq 0 then
        begin CompareNatFunc \leftarrow lInt; exit end;
     for i \leftarrow 0 to Count - 1 do
        begin lInt \leftarrow CompareInt(Items \uparrow [i].X, NatFuncPtr(aKey2) \uparrow .Items \uparrow [i].X);
        if lInt \neq 0 then
          begin CompareNatFunc \leftarrow lInt; exit end:
        lInt \leftarrow CompareInt(Items \uparrow [i].Y, NatFuncPtr(aKey2) \uparrow .Items \uparrow [i].Y);
        if lInt \neq 0 then
          begin CompareNatFunc \leftarrow lInt; exit end;
        end;
     end:
  CompareNatFunc \leftarrow 0;
  end;
```

391. Let $f: \mathbb{N} \to \mathbb{N}$ and $g: \mathbb{N} \to \mathbb{N}$ be partial functions. We say that f is "weaker" than g when $||f|| \le ||g||$ and for each $x \in \text{dom}(f)$ we have f(x) = g(x). If there is some $x \in \text{dom}(f)$ such that $x \notin \text{dom}(g)$, then f is not weaker than g.

If there is some $x \in \text{dom}(f)$ such that $x \in \text{dom}(g)$ and $f(x) \neq g(x)$, then f is not weaker than g.

function NatFunc. $WeakerThan(\mathbf{const}\ fNatFunc$: NatFunc): boolean; $\mathbf{var}\ i, k$: integer; begin $WeakerThan \leftarrow false$; if $Count \leq fNatFunc$. $Count\ \mathbf{then}$ begin for $k \leftarrow 0\ \mathbf{to}\ Count - 1\ \mathbf{do}$ begin $i \leftarrow Items \uparrow [k].X$; if $\neg fNatFunc$. $HasInDom(i)\ \mathbf{then}\ exit$; if $Items \uparrow [k].Y \neq fNatFunc$. $Value(i)\ \mathbf{then}\ exit$; end; $WeakerThan \leftarrow true$; end;

392. Let $f: \mathbb{N} \to \mathbb{N}$ and $g: \mathbb{N} \to \mathbb{N}$ be partial functions. We will say that f is a "multiple" of g if $||g|| \le ||f||$ and for each $x \in \text{dom}(g)$ we have $x \in \text{dom}(f)$ and $g(x) \le f(x)$.

There was some commented code for this function, which I removed.

```
function NatFunc.IsMultipleOf (const fNatFunc: NatFunc): boolean; var k, l: integer; begin IsMultipleOf \leftarrow false; if fNatFunc.Count \leq Count then begin for k \leftarrow 0 to fNatFunc.Count - 1 do
    if \neg HasInDom(fNatFunc.Items \uparrow [k].X) then exit else if Value(fNatFunc.Items \uparrow [k].X) < fNatFunc.Items \uparrow [k].Y then exit; IsMultipleOf \leftarrow true; end; end;
```

393. Comparing partial functions. Let $f: \mathbf{N} \to \mathbf{N}$ and $g: \mathbf{N} \to \mathbf{N}$ be partial functions.

If $||f|| \le ||g||$, for each $x \in \text{dom}(f)$ if $x \notin \text{dom}(g)$, then return 0. If $f(x) \ne g(x)$, then return 0. Otherwise return -1.

Else if $||g|| \le ||f||$, for each $x \in \text{dom}(g)$ if $x \notin \text{dom}(f)$, then return 0. If $f(x) \ne g(x)$, then return 0. Otherwise return +1.

This is difficult for me to grasp. It does not seem to adequately satisfy compare(f,g) = -compare(g,f), which is catastrophic. It is also unclear to me that this is transitive or reflexive. So it seems like it has no desirable properties.

I am confused why there is this function and also another similarly named function (§390).

The comment in Polish translates as, "Using WeakerThan you can shorten CompareWith!!!" At least, according to Google, that's the translation.

```
{ Uzywajac WeakerThan mozna skrocic CompareWith !!! }
function NatFunc.CompareWith(const fNatFunc: NatFunc): integer;
  var i, k: integer;
  begin Compare With \leftarrow 0;
  if Count \leq fNatFunc.Count then
     begin for k \leftarrow 0 to Count - 1 do
       begin i \leftarrow Items \uparrow [k].X;
       if \neg fNatFunc.HasInDom(i) then exit;
       if Items \uparrow [k].Y \neq fNatFunc.Value(i) then exit;
       end;
     Compare With \leftarrow -1; exit;
     end;
  if fNatFunc.Count < Count then
     begin for k \leftarrow 0 to fNatFunc.Count - 1 do
       begin i \leftarrow fNatFunc.Items \uparrow [k].X;
       if \neg HasInDom(i) then exit;
       if fNatFunc.Items \uparrow [k].Y \neq Value(i) then exit;
       end;
     Compare With \leftarrow 1; exit;
     end;
  end;
```

394. Let $f: \mathbb{N} \to \mathbb{N}$ and $g: \mathbb{N} \to \mathbb{N}$ be partial functions. Then we define $f + g: \mathbb{N} \to \mathbb{N}$ to be the partial function defined on $dom(f+g) = dom(f) \cup dom(g)$ such that for each $x \in dom(f \cap g)$ we have (f+g)(x) = f(x) + g(x), and for each $x \in dom(g) \setminus dom(g)$ we have (f+g)(x) = f(x), and for each $x \in dom(g) \setminus dom(f)$ we have (f+g)(x) = g(x).

There is some subtlety in the implementation because we have to check for overflows, i.e., when

$$g(x) \ge High(integer) - f(x)$$

```
for each x \in dom(f) \cap dom(g).
procedure NatFunc.Add(const aFunc: NatFunc);
  var k, l: integer;
  begin l \leftarrow 0;
  for k \leftarrow 0 to aFunc.Count - 1 do
     begin while (l < Count) \land (Items \uparrow [l].X < aFunc.Items \uparrow [k].X) do inc(l);
     if (l < Count) \land (Items \uparrow [l].X = aFunc.Items \uparrow [k].X) then
       begin if (Has overflow occurred in NatFunc.Add? 395) then RunError(215);
        inc(Items\uparrow[l].Y, aFunc.Items\uparrow[k].Y);
       end
     else AtInsert(l, aFunc.Items \uparrow [k]);
     end;
  end;
        An overflow occurs if f(x) + g(x) is greater than High(integer) (the maximum value for an integer).
\langle Has overflow occurred in NatFunc.Add? 395\rangle \equiv
  Items \uparrow [l].Y > (High(integer) - aFunc.Items \uparrow [k].Y)
This code is used in section 394.
```

396. Sum values of partial function. For a partial function $f: \mathbb{N} \to \mathbb{N}$, we have

$$CountAll(f) = \sum_{n \in dom(f)} f(n).$$

```
\langle NatFunc \text{ implementation } 380 \rangle +\equiv
function NatFunc.CountAll: integer;
var k,l: integer;
begin l \leftarrow 0;
for k \leftarrow 0 to Count - 1 do inc(l, Items \uparrow [k].Y);
CountAll \leftarrow l;
end;
```

Section 9.18. SEQUENCES OF NATURAL NUMBERS

```
397.
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
  NatSeq = \mathbf{object} \ (NatFunc)
     constructor InitNatSeq(ALimit, ADelta: integer);
     procedure InsertElem(X : integer); virtual;
     function Value(fElem: integer): integer; virtual;
     function IndexOf(Y:integer):integer;
  end;
398. Constructor.
\langle \text{NatSeq implementation 398} \rangle \equiv
constructor NatSeq.InitNatSeq(ALimit, ADelta: integer);
  begin inherited\ Init(ALimit, ADelta);\ nConsistent \leftarrow true;
  end:
This code is used in section 183.
399. If we have a finite sequence (a_0, \ldots, a_{n-1}), then inserting an element x into it will yield the finite
sequence (a_0,\ldots,a_{n-1},x).
procedure NatSeq.InsertElem(X:integer);
  var lPair: IntPair;
  begin lPair.X \leftarrow Count; \ lPair.Y \leftarrow X; \ inherited Insert(lPair);
400. The value for the k^{th} element in a sequence (a_0, \ldots, a_{n-1}) is a_k when 0 \le k < n, and we take it to
be 0 otherwise.
function NatSeq. Value (fElem: integer): integer;
     begin
         \{(0) = \text{ind} \text{ and } \}
     (fElem < count) then Value \leftarrow Items \uparrow [fElem].Y
  else Value \leftarrow 0;
     end;
401.
       The index for a_i in the sequence (a_0, \ldots, a_{n-1}) is i when a_i is in the sequence. Otherwise, we return
function NatSeq.IndexOf(Y:integer):integer;
  var lResult: integer;
  begin for lResult \leftarrow Count - 1 downto 0 do
    if Items \uparrow [lResult].Y = Y then
       begin IndexOf \leftarrow lResult; exit
       end;
  IndexOf \leftarrow -1;
  end;
```

124 INTEGER SEQUENCES Mizar Parser $\S402$

Section 9.19. INTEGER SEQUENCES

```
402.
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
  IntegerListPtr = \uparrow IntegerList;
  IntegerList = array [0...MaxIntegerListSize - 1] of integer;
  PIntSequence = \uparrow IntSequence;
  IntSequencePtr = PIntSequence;
  IntSequence = \mathbf{object} \ (MObject)
    fList: IntegerListPtr;
     fCount: integer;
    fCapacity: integer;
     constructor Init(aCapacity : integer);
     constructor CopySequence(const aSeq: IntSequence);
     constructor MoveSequence(var aSeq : IntSequence);
     destructor Done; virtual;
     procedure IntListError(Code, Info: integer); virtual;
     procedure SetCapacity(aCapacity: integer); virtual;
     procedure Clear; virtual;
     function Insert(aInt : integer): integer; virtual;
     procedure AddSequence(const aSeq: IntSequence); virtual;
     function IndexOf (aInt : integer): integer; virtual;
     procedure AtDelete(aIndex:integer); virtual;
     function Value(aIndex: integer): integer; virtual;
     procedure AtInsert(aIndex, aInt : integer); virtual;
     procedure AtPut(aIndex, aInt: integer); virtual;
  end;
403. We will need to quicksort lists of integers. This will mutate the aList argument, making it sorted.
See also \S 257 and \S 334.
  This procedure does not appear to be used anywhere in Mizar.
\langle IntSequence implementation 403 \rangle \equiv
     { integer Sequences & Sets }
procedure IntQuickSort(aList : IntegerListPtr; L, R : integer);
  var I, J, P, lTemp: integer;
  begin repeat I \leftarrow L; J \leftarrow R; P \leftarrow aList \uparrow [(L+R) \text{ shr } 1];
     repeat while CompareInt(aList\uparrow[I], P) < 0 do inc(I);
       while CompareInt(aList\uparrow[J], P) > 0 do Dec(J);
       if I \leq J then
         begin lTemp \leftarrow aList\uparrow[I]; \ aList\uparrow[I] \leftarrow aList\uparrow[J]; \ aList\uparrow[J] \leftarrow lTemp; \ inc(I); \ Dec(J);
         end:
     until I > J;
    if L < J then IntQuickSort(aList, L, J);
     L \leftarrow I;
  until I \geq R;
  end:
This code is used in section 183.
```

§404 Mizar Parser INTEGER SEQUENCES 125

```
Constructor. We can create an empty sequence of integers, with a given capacity.
constructor IntSequence.Init(aCapacity: integer);
  begin inherited Init; fList \leftarrow nil; fCount \leftarrow 0; fCapacity \leftarrow 0; SetCapacity(aCapacity);
  end:
405.
       Copy constructor. We can copy an existing sequence.
constructor IntSequence.CopySequence(const aSeq: IntSequence);
  begin Init(aSeq.fCapacity); AddSequence(aSeq);
  end:
406.
       Move constructor. We can create a new array in heap, and move all the elements from a given
sequence over, then free up the given sequence.
constructor IntSequence.MoveSequence(var aSeq: IntSequence);
  begin inherited Init; fCount \leftarrow aSeq.fCount; fCapacity \leftarrow aSeq.fCapacity; fList \leftarrow aSeq.fList;
  aSeq.fCount \leftarrow 0; aSeq.fCapacity \leftarrow 0; aSeq.fList \leftarrow nil;
  end:
407. Destructor.
destructor IntSequence.Done;
  begin inherited Done; fCount \leftarrow 0; SetCapacity(0);
  end:
       Appending an element. Given a finite sequence of integers (a_0, \ldots, a_{n-1}), we can append a value
408.
x to produce the finite sequence (a_0, \ldots, a_{n-1}, x). This will mutate the caller.
function IntSequence.Insert(aInt:integer): integer;
  begin if fCount = fCapacity then SetCapacity(fCapacity + GrowLimit(fCapacity));
  fList \uparrow [fCount] \leftarrow aInt; Insert \leftarrow fCount; inc(fCount);
  end:
409. Appending a sequence. This takes a finite sequence (a_0, \ldots, a_{n-1}) and another finite sequence
(b_0,\ldots,b_{m-1}), then forms a new finite sequence (a_0,\ldots,a_{n-1},b_0,\ldots,b_{m-1}). It mutates the caller.
procedure IntSequence.AddSequence(const aSeq: IntSequence);
  var I, r: integer;
  begin for I \leftarrow 0 to aSeq.fCount - 1 do r \leftarrow Insert(aSeq.fList \uparrow [I]);
  end;
410. Clearing a sequence.
procedure IntSequence.Clear;
  begin if fCount \neq 0 then
    begin fCount \leftarrow 0; SetCapacity(0);
    end:
  end;
411. Delete entry in sequence. Removing the i^{th} entry in the sequence (a_0, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_{n-1})
yields the finite sequence (a_0, \ldots, a_{i-1}, a_{i+1}, \ldots, a_{n-1}). If i < 0 or n-1 < i, then we raise an error.
procedure IntSequence.AtDelete(aIndex: integer);
  begin if (aIndex < 0) \lor (aIndex \ge fCount) then IntListError(coIndexError, aIndex);
  if aIndex < fCount then Move(fList \uparrow [aIndex + 1], fList \uparrow [aIndex], (fCount - aIndex) * SizeOf(integer));
  end;
```

126

end:

```
412.
       We report errors using this helper function.
procedure IntSequence.IntListError(Code, Info: integer);
  begin RunError(212 - Code); {! poprawic bledy}
  end:
413. Let (a_0, \ldots, a_{n-1}) be a finite sequence. The value at index i is a_i when 0 \le i \le n-1, otherwise it
raises an error.
function IntSequence. Value (aIndex: integer): integer:
  begin if (aIndex < 0) \lor (aIndex \ge fCount) then IntListError(coIndexError, aIndex);
  Value \leftarrow fList \uparrow [aIndex];
  end;
414. For a finite sequence (a_0,\ldots,a_{n-1}) and a value x, if there is some entry a_i=x with a_i\neq x for i< i,
then return i. Otherwise return -1.
function IntSequence.IndexOf(aInt:integer): integer;
  var lResult: integer;
  begin for lResult \leftarrow fCount - 1 downto 0 do
    if fList \uparrow [lResult] = aInt then
       begin IndexOf \leftarrow lResult; exit
       end;
  IndexOf \leftarrow -1;
  end;
415. Given a finite sequence (a_0, \ldots, a_{n-1}), an index i, and a value x:
(1) If i < 0 or i is too big, raise an error.
(2) If the logical size of the sequence equals its capacity, then grow the underlying array.
(3) If i is less than the logical size i < n-1, then shift all the entries to the right by 1 so we have
     (a_0,\ldots,a_{i-1},0,a_i,\ldots,a_{n-1})
(4) Set the i^{th} entry to x, so we end up with the caller becoming (a_0, \ldots, a_{i-1}, x, a_i, \ldots, a_{n-1}).
procedure IntSequence.AtInsert(aIndex, aInt: integer);
  begin if (aIndex < 0) \lor (aIndex > fCount) then IntListError(coIndexError, aIndex);
  if fCount = fCapacity then SetCapacity(fCapacity + GrowLimit(fCapacity));
  if aIndex < fCount then Move(fList \uparrow [aIndex], fList \uparrow [aIndex + 1], (fCount - aIndex) * SizeOf(integer));
  fList \uparrow [aIndex] \leftarrow aInt; inc(fCount);
  end;
416. Update entry of sequence. For a sequence (a_0, \ldots, a_{n-1}), an index i, and a new value x, if
0 \le i \le n-1 then we set a_i \leftarrow x. Otherwise we have the index be out of bounds (0 < i \text{ or } n-1 < i), and
we should raise an error.
procedure IntSequence.AtPut(aIndex, aInt: integer);
  begin if (aIndex < 0) \lor (aIndex \ge fCount) then IntListError(coIndexError, aIndex);
  fList \uparrow [aIndex] \leftarrow aInt;
```

 $\S417$ Mizar Parser INTEGER SEQUENCES 127

417. Grow the underlying array. When we want to increase (or decrease) the capacity of the underlying array, we invoke this function. It will copy over the relevant contents.

```
procedure IntSequence.SetCapacity(aCapacity:integer);
var lList: IntegerListPtr;
begin if aCapacity < fCount then aCapacity \leftarrow fCount;
if aCapacity > MaxListSize then aCapacity \leftarrow MaxListSize;
if aCapacity \neq fCapacity then
begin if aCapacity = 0 then lList \leftarrow nil
else begin GetMem(lList, aCapacity * SizeOf(integer));
if (fCount \neq 0) \land (fList \neq nil) then Move(fList \uparrow, lList \uparrow, fCount * SizeOf(integer));
end;
if fCapacity \neq 0 then FreeMem(fList, fCapacity * SizeOf(integer));
fList \leftarrow lList; fCapacity \leftarrow aCapacity;
end;
end;
```

128 INTEGER SETS Mizar Parser $\S418$

Section 9.20. INTEGER SETS

end;

```
418.

⟨ Public interface for mobjects.pas 184⟩ +≡

PIntSet = ↑IntSet;

IntSetPtr = pIntSet;

IntSet = object (IntSequence)

function Insert(aInt: integer): integer; virtual;

function DeleteInt(aInt: integer): integer; virtual;

function Find(aInt: integer; var aIndex: integer): boolean; virtual;

function IndexOf(aInt: integer): integer; virtual;

procedure AtInsert(aIndex, aInt: integer); virtual;

function IsInSet(aInt: integer): boolean; virtual;

function IsSqualTo(const aSet: IntSet): boolean; virtual;

function IsSubsetOf(const aSet: IntSet): boolean; virtual;

function IsSupersetOf(var aSet: IntSet): boolean; virtual;

function Misses(var aSet: IntSet): boolean; virtual;
```

419. When inserting an element x into a set A, we check if $x \in A$ is already a member. If so, then we're done.

Otherwise, we ensure the capacity of the set can handle adding another element. Then we shift all elements greater than x over to the right by 1. We finally insert x into the underlying array.

```
⟨ IntSet Implementation 419⟩ ≡ function IntSet.Insert(aInt: integer): integer; var lIndex: integer; begin if Find(aInt, lIndex) then { already contains the element? } begin Insert ← lIndex; exit end; if fCount = fCapacity then SetCapacity(fCapacity + GrowLimit(fCapacity)); if lIndex < fCount then Move(fList \uparrow [lIndex], fList \uparrow [lIndex + 1], (fCount - lIndex) * SizeOf(integer)); fList \uparrow [lIndex] ← aInt; inc(fCount); Insert ← lIndex; end;

This code is used in section 183.
```

420. Removing an element from a set. This will return the former index of the element in the underlying array.

```
function IntSet.DeleteInt(aInt:integer): integer;

var lIndex:integer;

begin DeleteInt \leftarrow -1;

if Find(aInt, lIndex) then

begin DeleteInt \leftarrow lIndex; AtDelete(lIndex) end

end;
```

§421 Mizar Parser INTEGER SETS 129

421. We can use bisection search to find an element aInt in the underlying array. It will mutate aIndex to be where the entry should be, and return true if the element is a member of the set (and false otherwise).

```
function IntSet.Find(aInt: integer; var aIndex: integer): boolean;
  var L, H, I, C: integer;
  begin Find \leftarrow False; L \leftarrow 0; H \leftarrow fCount - 1;
  while L \leq H do
     begin I \leftarrow (L+H) shr 1; C \leftarrow CompareInt(fList \uparrow [I], aInt);
     if C < 0 then L \leftarrow I + 1
     else begin H \leftarrow I - 1;
       if C = 0 then
         begin Find \leftarrow True; L \leftarrow I; end;
       end:
     end;
  aIndex \leftarrow L;
  end;
       We can find the index of an element (if it is present) by using bisection search.
function IntSet.IndexOf(aInt:integer): integer;
  var lResult: integer;
  begin if \neg Find(aInt, lResult) then lResult \leftarrow -1;
  IndexOf \leftarrow lResult;
  end;
423. The AtInsert method is "grandfathered in", but not supported, so we raise an error if anyone tries
procedure IntSet.AtInsert(aIndex, aInt: integer);
  begin IntListError(coSortedListError, 0);
  end:
       We can test if an integer is an element of the set, again just piggie-backing off bisection search.
function IntSet.IsInSet(aInt: integer): boolean;
  var I: integer;
  begin IsInSet \leftarrow Find(aInt, I);
  end;
425. Testing if two finite sets A and B of integers are equal requires |A| = |B| and for each x \in A we have
x \in B. If these conditions are not both met, then A \neq B.
function IntSet.IsEqualTo(const aSet: IntSet): boolean;
  var I: integer;
  begin IsEqualTo \leftarrow false;
  if fCount \neq aSet.fCount then exit;
  for I \leftarrow 0 to fCount - 1 do
     if fList \uparrow [I] \neq aSet.fList \uparrow [I] then exit;
  IsEqualTo \leftarrow true;
  end:
```

130 INTEGER SETS Mizar Parser $\S426$

```
Subset predicate. We can test A \subseteq B by |A| \le |B| and for each a \in A we have a \in B.
function IntSet.IsSubsetOf(const aSet: IntSet): boolean;
  var i, j, lInt: integer;
  begin IsSubsetOf \leftarrow false;
  if aSet.fCount < fCount then exit;
  j \leftarrow 0; { index of B }
  for i \leftarrow 0 to fCount - 1 do { loop over a \in A }
     begin lInt \leftarrow fList \uparrow [i];
     while (j < aSet.fCount) \land (aSet.fList \uparrow [j] < lInt) do inc(j);
     if (j = aSet.fCount) \lor (aSet.fList\uparrow[j] \neq fList\uparrow[i]) then exit;
     end;
  IsSubsetOf \leftarrow true;
  end;
427. Superset predicate. We have A \supset B if B \subseteq A.
function IntSet.IsSupersetOf(var aSet : IntSet): boolean;
  begin IsSupersetOf \leftarrow aSet.IsSubsetOf(Self);
  end;
428.
      Test for disjointness. We have A \cap B = \emptyset if every a \in A is such that a \notin B.
function IntSet. Misses (var aSet : IntSet): boolean;
  var k: integer;
  begin if fCount > aSet.fCount then
     begin for k \leftarrow 0 to aSet.fCount - 1 do
       if IsInSet(aSet.fList\uparrow[k]) then
          begin Misses \leftarrow false; exit end
     end
  else begin for k \leftarrow 0 to fCount - 1 do
       if aSet.IsInSet(fList\uparrow[k]) then
          begin Misses \leftarrow false; exit end;
     end;
  Misses \leftarrow true;
  end;
```

Section 9.21. PARTIAL BINARY INTEGER FUNCTIONS

```
429. We want to describe partial functions like f: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}. These are encoded as finite sets of triples
\{(x,y,f(x,y))\in \mathbf{Z}\times\mathbf{Z}\times\mathbf{Z}\}. So we need to introduce triples of integers.
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
  IntTripletListPtr = \uparrow IntTripletList;
  IntTripletList = array [0 ... MaxIntTripletSize - 1] of IntTriplet;
  BinIntFuncPtr = \uparrow BinIntFunc;
  BinIntFunc = \mathbf{object} \ (MObject)
    fList: IntTripletListPtr;
    fCount: integer;
    fCapacity: integer;
    constructor Init(aLimit : integer);
    procedure BinIntFuncError(aCode, aInfo: integer); virtual;
    destructor Done; virtual;
    procedure Insert(const aItem: IntTriplet); virtual;
    procedure AtDelete(aIndex:integer);
    procedure SetCapacity(aLimit : integer); virtual;
    procedure DeleteAll;
    function Search (X1, X2 : integer; var aIndex : integer): boolean; virtual;
    function IndexOf(X1, X2 : integer): integer;
    constructor CopyBinIntFunc(var aFunc : BinIntFunc);
    function HasInDom(X1, X2 : integer): boolean; virtual;
    procedure Assign(X1, X2, Y : integer); virtual;
    procedure Up(X1, X2 : integer); virtual;
    procedure Down(X1, X2 : integer); virtual;
    function Value(X1, X2 : integer): integer; virtual;
    procedure Add(const aFunc: BinIntFunc); virtual;
    function CountAll: integer; virtual;
  end;
430. We have a convenience function for reporting errors.
\langle \text{ Partial Binary integer Functions } 430 \rangle \equiv
procedure BinIntFunc.BinIntFuncError(aCode, aInfo: integer);
  begin RunError(212 - aCode); end;
This code is used in section 183.
       Constructor. We initialize the empty partial function.
constructor BinIntFunc.Init(aLimit:integer);
  begin MObject.Init; fList \leftarrow nil; fCount \leftarrow 0; fCapacity \leftarrow 0; SetCapacity(aLimit);
  end:
432. Destructor.
destructor BinIntFunc.Done:
  begin fCount \leftarrow 0; SetCapacity(0);
  end;
```

433. If we have a partial function $f: \mathbf{Z} \times \mathbf{Z} \to \mathbf{Z}$ and a triple (x, y, z), then check if $(x, y) \in \text{dom}(f)$. If so, we're done. Otherwise we add f(x,y) = z to the partial function. **procedure** BinIntFunc.Insert(**const** aItem: IntTriplet); **var** *I*: integer; begin if $\neg Search(aItem.X1, aItem.X2, I)$ then begin if $(I < 0) \lor (I > fCount)$ then **begin** BinIntFuncError(coIndexError, 0); exit; end; if fCapacity = fCount then SetCapacity(fCapacity + GrowLimit(fCapacity));if $I \neq fCount$ then $Move(fList \uparrow [I], fList \uparrow [I+1], (fCount-I) * SizeOf(IntTriplet));$ $fList \uparrow [I] \leftarrow aItem; inc(fCount);$ end: end; **434.** Given $f: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, we represent it as an array of $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$. So we can remove the entry at index i when $0 \le i < ||f||$. Otherwise when i < 0 or $||f|| \le i$, raise an error. procedure BinIntFunc.AtDelete(aIndex : integer); **var** *i*: *integer*; **begin if** $(aIndex < 0) \lor (aIndex > fCount)$ **then begin** BinIntFuncError(coIndexError, 0); exit; end; if aIndex < fCount - 1 then for $i \leftarrow aIndex$ to fCount - 2 do $fList \uparrow [i] \leftarrow fList \uparrow [i+1];$ Dec(fCount);end; 435. Ensure capacity. **procedure** BinIntFunc.SetCapacity(aLimit:integer); **var** aItems: IntTripletListPtr; **begin if** aLimit < fCount **then** $aLimit \leftarrow fCount$; if aLimit > MaxIntTripletSize then $ALimit \leftarrow MaxIntTripletSize$; if $aLimit \neq fCapacity$ then begin if ALimit = 0 then $AItems \leftarrow nil$ else begin GetMem(AItems, ALimit * SizeOf(IntTriplet));if $(fCount \neq 0) \land (fList \neq nil)$ then $Move(fList \uparrow, aItems \uparrow, fCount * SizeOf(IntTriplet));$ end; if $fCapacity \neq 0$ then FreeMem(fList, fCapacity * SizeOf(IntTriplet)); $fList \leftarrow aItems; fCapacity \leftarrow aLimit;$ end: end; **436.** Deleting all entries in a partial function $\mathbf{Z} \times \mathbf{Z} \to \mathbf{Z}$ amounts to setting the logical size of the underlying

436. Deleting all entries in a partial function $\mathbf{Z} \times \mathbf{Z} \to \mathbf{Z}$ amounts to setting the logical size of the underlying dynamic array to zero.

```
procedure BinIntFunc.DeleteAll;
begin fCount \leftarrow 0; end;
```

```
We can use bisection search to find an entry (x_1, x_2) such that (x_1, x_2) \in \text{dom}(f).
function BinIntFunc.Search(X1, X2 : integer; var aIndex : integer): boolean;
  var L, H, I, C: integer;
  begin Search \leftarrow False; L \leftarrow 0; H \leftarrow fCount - 1;
  while L \leq H do
     begin I \leftarrow (L+H) shr 1; C \leftarrow CompareIntPairs(fList\uparrow[I].X1, fList\uparrow[I].X2, X1, X2);
     if C < 0 then L \leftarrow I + 1
     else begin H \leftarrow I - 1;
       if C = 0 then
          begin Search \leftarrow True; L \leftarrow I; end;
       end;
     end;
  aIndex \leftarrow L;
  end;
        Copy constructor. This leaves aFunc unchanged, and clones aFunc.
constructor BinIntFunc.CopyBinIntFunc(var aFunc : BinIntFunc);
  begin Init(aFunc.fCapacity); Move(aFunc.fList\uparrow, fList\uparrow, aFunc.fCapacity * SizeOf(IntTriplet));
  fCount \leftarrow aFunc.fCount;
  end;
439. Given f: \mathbf{Z} \times \mathbf{Z} \rightharpoonup \mathbf{Z} and (x_1, x_2), find the index for the underlying dynamic array i such that it
contains (x_1, x_2, f(x_1, x_2)). If there is no such entry, i = -1 is returned.
function BinIntFunc.IndexOf(X1, X2 : integer): integer;
  var I: integer;
  begin IndexOf \leftarrow -1;
  if Search(X1, X2, I) then IndexOf \leftarrow I;
440.
       Test if (x_1, x_2) \in \text{dom}(f).
function BinIntFunc.HasInDom(X1, X2 : integer): boolean;
  var I: integer;
  begin HasInDom \leftarrow Search(X1, X2, I);
  end;
441. Given f: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}, and (x_1, x_2) \in \mathbb{Z} \times \mathbb{Z} and y \in \mathbb{Z}, try setting f(x_1, x_2) = y provided
(x_1, x_2) \notin \text{dom}(f) or if (x_1, x_2, y) \in f already. If f(x_1, x_2) \neq y already exists, then raise an error.
procedure BinIntFunc.Assign(X1, X2, Y : integer);
  var lIntTriplet: IntTriplet;
  begin if HasInDom(X1, X2) \wedge (Value(X1, X2) \neq Y) then
     begin BinIntFuncError(coDuplicate, 0); exit
  lIntTriplet.X1 \leftarrow X1; lIntTriplet.X2 \leftarrow X2; lIntTriplet.Y \leftarrow Y; Insert(lIntTriplet);
  end;
```

 $CountAll \leftarrow l;$

end;

```
Given f: \mathbf{Z} \times \mathbf{Z} \to \mathbf{Z} and (x_1, x_2) \in \mathbf{Z} \times \mathbf{Z}. If (x_1, x_2) \in \text{dom}(f), then set f(x_1, x_2) \leftarrow f(x_1, x_2) + 1.
Otherwise set f(x_1, x_2) \leftarrow 1.
procedure BinIntFunc. Up(X1, X2 : integer);
   var I: integer; lIntTriplet: IntTriplet;
   begin if Search(X1, X2, I) then inc(fList \uparrow [I].Y)
   else begin lIntTriplet.X1 \leftarrow X1; lIntTriplet.X2 \leftarrow X2; lIntTriplet.Y \leftarrow 1; Insert(lIntTriplet);
     end;
   end;
443. Given f: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} and (x_1, x_2) \in \mathbb{Z} \times \mathbb{Z}. If (x_1, x_2) \in \text{dom}(f), then set f(x_1, x_2) \leftarrow f(x_1, x_2) - 1.
Further, if f(x_1, x_2) = 0, then remove it from the underlying dynamic array.
   Otherwise for (x_1, x_2) \notin \text{dom}(f), raise an error.
procedure BinIntFunc.Down(X1, X2 : integer);
   var I: integer;
   begin if Search(X1, X2, I) then
     begin dec(fList\uparrow[I].Y);
     if fList \uparrow [I].Y = 0 then AtDelete(I);
   else BinIntFuncError(coConsistentError, 0);
   end;
444. Given f: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}, and (x_1, x_2) \in \mathbb{Z} \times \mathbb{Z}, if (x_1, x_2) \notin \text{dom}(f) then raise an error. Otherwise when
(x_1, x_2) \in \text{dom}(f), return f(x_1, x_2).
function BinIntFunc.Value(X1, X2 : integer): integer;
   var I: integer;
   begin if Search(X1, X2, I) then Value \leftarrow fList \uparrow [I].Y
   else BinIntFuncError(coDuplicate, 0);
   end;
445. Given two partial functions f, g: \mathbf{Z} \times \mathbf{Z} \to \mathbf{Z}, compute f + g: \mathbf{Z} \times \mathbf{Z} \to \mathbf{Z}. This is defines by:
(1) For (x_1, x_2) \in \text{dom}(f) \cap \text{dom}(g), set (f+g)(x_1, x_2) = f(x_1, x_2) + g(x_1, x_2)
(2) For (x_1, x_2) \in \text{dom}(f) \setminus \text{dom}(g), set (f + g)(x_1, x_2) = f(x_1, x_2)
(3) For (x_1, x_2) \in \text{dom}(g) \setminus \text{dom}(f), set (f + g)(x_1, x_2) = g(x_1, x_2).
     { TODO: this is inefficient, since the search is repeated in the Assign method; fix this both here and
        in other similar methods }
procedure BinIntFunc.Add(const aFunc: BinIntFunc);
   var k, l: integer;
   begin for k \leftarrow 0 to aFunc.fCount - 1 do
     if Search(aFunc.fList\uparrow[k].X1, aFunc.fList\uparrow[k].X2, l) then inc(fList\uparrow[l].Y, aFunc.fList\uparrow[k].Y)
     else Assign(aFunc.fList\uparrow[k].X1, aFunc.fList\uparrow[k].X2, aFunc.fList\uparrow[k].Y);
   end;
446. Sum. For f: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}, we compute
                                             CountAll(f) = \sum_{(m,n) \in dom(f)} f(m,n).
function BinIntFunc.CountAll: integer;
   var k, l: integer;
   begin l \leftarrow 0;
   for k \leftarrow 0 to fCount - 1 do inc(l, fList \uparrow [k].Y);
```

Section 9.22. PARTIAL INTEGERS TO PAIR OF INTEGERS FUNCTIONS

```
447.
\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv
  Int2PairOfInt = \mathbf{record} \ X, Y1, Y2: integer;
  Int2PairOfIntFuncPtr = \uparrow Int2PairOfIntFunc;
  Int2PairOfIntFunc = object (MObject)
    fList: array of Int2PairOfInt;
    fCount: integer;
    fCapacity: integer;
    constructor Init(aLimit : integer);
    procedure Int2PairOfIntFuncError(aCode, aInfo: integer); virtual;
    destructor Done; virtual;
    procedure Insert(const aItem: Int2PairOfInt); virtual;
    procedure AtDelete(aIndex : integer);
    procedure SetCapacity(aLimit:integer); virtual;
    procedure DeleteAll;
    function Search(X : integer; var aIndex : integer): boolean; virtual;
    function IndexOf(X:integer): integer;
    constructor CopyInt2PairOfIntFunc(var aFunc : Int2PairOfIntFunc);
    function HasInDom(X:integer): boolean; virtual;
    procedure Assign(X, Y1, Y2 : integer); virtual;
    function Value(X:integer): IntPair; virtual;
  end;
448. We have a helper function for raising errors.
\langle Partial integers to Pair of integers Functions 448\rangle \equiv
    { Partial integers to Pair of integers Functions }
procedure Int2PairOfIntFunc.Int2PairOfIntFuncError(aCode, aInfo: integer);
  begin RunError(212 - aCode);
  end:
This code is used in section 183.
      Constructor. Creates an empty f: \mathbf{Z} \to \mathbf{Z} \times \mathbf{Z} with an underlying dynamic array whose capacity is
given as the argument aLimit.
constructor Int2PairOfIntFunc.Init(aLimit : integer);
  begin MObject.Init; fList \leftarrow \mathbf{nil}; fCount \leftarrow 0; fCapacity \leftarrow 0; SetCapacity(aLimit);
  end;
450. Destructor.
destructor Int2PairOfIntFunc.Done;
  begin fCount \leftarrow 0; SetCapacity(0);
  end:
```

136

451. Inserting (x, y_1, y_2) into $f: \mathbf{Z} \to \mathbf{Z} \times \mathbf{Z}$ amounts to checking if $(x, y_1, y_2) \in f$. If not, then insert the entry. Otherwise, if $(x, y_1, y_2) \notin f$ but $x \in \text{dom}(f)$, then raise an error. Otherwise do nothing. **procedure** Int2PairOfIntFunc.Insert(**const** aItem: Int2PairOfInt); **var** *I*: integer; begin if $\neg Search(aItem.X, I)$ then begin if $(I < 0) \lor (I > fCount)$ then **begin** Int2PairOfIntFuncError(coIndexError, 0); exit; end; if fCapacity = fCount then SetCapacity(fCapacity + GrowLimit(fCapacity));if $I \neq fCount$ then Move(fList[I], fList[I+1], (fCount-I) * SizeOf(Int2PairOfInt)); $fList[I] \leftarrow aItem; inc(fCount);$ end else if $(fList[I]. Y1 \neq aItem. Y1) \lor (fList[I]. Y2 \neq aItem. Y2)$ then **begin** Int2PairOfIntFuncError(coDuplicate, 0); exit; end; end: 452.Delete an entry from the underlying dynamic array. Raise an error if the index given is out of bounds. **procedure** Int2PairOfIntFunc.AtDelete(aIndex:integer); **var** *i*: *integer*; **begin if** $(aIndex < 0) \lor (aIndex > fCount)$ **then begin** Int2PairOfIntFuncError(coIndexError, 0); exit; end; if aIndex < fCount - 1 then for $i \leftarrow aIndex$ to fCount - 2 do $fList[i] \leftarrow fList[i+1]$; dec(fCount);end; **453**. **procedure** Int2PairOfIntFunc.SetCapacity(aLimit:integer); **begin if** aLimit < fCount **then** $aLimit \leftarrow fCount$; $setlength(fList, aLimit); fCapacity \leftarrow aLimit;$ end; We can "soft delete" all entries in the partial function. **procedure** Int2PairOfIntFunc.DeleteAll; **begin** $fCount \leftarrow 0$; end;

```
455.
        We can bisection search on the domain.
function Int2PairOfIntFunc.Search(X:integer; var aIndex:integer): boolean;
  var L, H, I, C: integer;
  begin Search \leftarrow False; L \leftarrow 0; H \leftarrow fCount - 1;
  while L \leq H do
     begin I \leftarrow (L+H) shr 1; C \leftarrow CompareInt(fList[I],X,X);
     if C < 0 then L \leftarrow I + 1
     else begin H \leftarrow I - 1;
       if C = 0 then
         begin Search \leftarrow True; L \leftarrow I;
          end:
       end;
     end;
  aIndex \leftarrow L;
  end;
456. Copy constructor. This leaves the argument aFunc unchanged.
constructor Int2PairOfIntFunc.CopyInt2PairOfIntFunc(var aFunc: Int2PairOfIntFunc);
  \mathbf{begin}\ Init(aFunc.fCapacity);\ Move(aFunc.fList[0],fList[0],aFunc.fCapacity*SizeOf(Int2PairOfInt));
  fCount \leftarrow aFunc.fCount;
  end;
457. Find the index in the underlying dynamic array for x \in \text{dom}(f). If x \notin \text{dom}(f), then return -1.
function Int2PairOfIntFunc.IndexOf(X:integer): integer;
  var I: integer;
  begin IndexOf \leftarrow -1;
  if Search(X, I) then IndexOf \leftarrow I;
  end:
458.
      Test if x \in \text{dom}(f).
function Int2PairOfIntFunc.HasInDom(X:integer): boolean;
  var I: integer;
  begin HasInDom \leftarrow Search(X, I);
  end;
459. Attempt to insert (x, y_1, y_2) into f: \mathbf{Z} \rightharpoonup \mathbf{Z} \times \mathbf{Z}.
procedure Int2PairOfIntFunc.Assign(X, Y1, Y2 : integer);
  var lInt2PairOfInt: Int2PairOfInt;
  begin lInt2PairOfInt.X \leftarrow X; lInt2PairOfInt.Y1 \leftarrow Y1; lInt2PairOfInt.Y2 \leftarrow Y2;
  Insert(lInt2PairOfInt);
  end;
460. Given f: \mathbf{Z} \to \mathbf{Z} \times \mathbf{Z} and x \in \mathbf{Z}, if x \in \text{dom}(f) return f(x). Otherwise raise an error.
function Int2PairOfIntFunc.Value(X:integer): IntPair;
  var I: integer;
  begin if Search(X, I) then
     begin Result.X \leftarrow fList[I].Y1; Result.Y \leftarrow fList[I].Y2;
  else Int2PairOfIntFuncError(coDuplicate, 0);
  end;
```

Mizar Parser

138

461. We have a myriad of random declarations, so we just stick them all here. $\langle \text{Public interface for mobjects.pas } 184 \rangle + \equiv$ { Comparing Strings wrt MStrObj } **function** CompareStringPtr(aKey1, aKey2 : Pointer): integer; $\{ \, {\rm Comparing \,\, Strings \,\, and \,\, integers} \, \}$ **function** CompareStr(aStr1, aStr2 : String): integer; **function** CompareIntPairs(X1, Y1, X2, Y2: Longint): integer; { Dynamic String handling routines } function NewStr(const S: String): PString; **procedure** DisposeStr(P : PString);**function** GrowLimit(aLimit:integer): integer; {Abstract notification procedure} **function** CompareNatFunc(aKey1, aKey2 : Pointer): integer; **procedure** Abstract1; $\mathbf{var}\ \mathit{EmptyNatFunc}\colon \mathit{NatFunc};$

8462 Mizar Parser XML DICTIONARY 139

File 10

XML Dictionary

462. We have several types declared in the xml_dict.parse file. These are enumerated types, and string constants for their names.

```
\langle \text{xml\_dict.pas } 462 \rangle \equiv
  ⟨GNU License 4⟩
unit xml_-dict;
  interface
  uses mobjects;
  { known (and only allowed) XML elements }
  type\ XMLElemKind = (elUnknown, elAdjective, elAdjectiveCluster, elArticleID, elAncestors,
        elArguments, elBlock, elConditions, elCorrectnessConditions, elDefiniens, elDirective, elEnviron,
        elEquality, elFieldSegment, elFormat, elFormats, elIdent, elIterativeStep, elLabel, elLink,
        elLoci, elLociEquality, elLocus, elNeqatedAdjective, elPartialDefiniens, elPriority, elProposition,
        elProvisionalFormulas, elRedefine, elRightCircumflexSymbol, elSchematicVariables, elScheme,
        elSelector, elSetMember, elSkippedProof, elSymbol, elSymbolCount, elSymbols, elSubstitution,
        elTypeSpecification, elTypeList, elVariable, elVariables, elVocabularies, elVocabulary);
  { known XML attributes }
    XMLAttrKind = (atUnknown, atAid, atArgNr, atArticleId, atArticleExt, atCol, atCondition,
        atConstrNr, atIdNr, atInfinitive, atKind, atLabelNr, atLeftArgNr, atLine, atMizfiles, atName,
        atNegated, atNr, atNumber, atOrigin, atPosLine, atPosCol, atPriority, atProperty,
        atRightSymbolNr, atSchNr, atSerialNr, atShape, atSpelling, atSymbolNr, atValue, atVarNr,
        atVarSort, atX, atX1, atX2, atY, atY1, atY2);
  const XMLElemName: array [XMLElemKind] of string = ('Unknown', 'Adjective',
           'Adjective-Cluster', 'ArticleID', 'Ancestors', 'Arguments', 'Block', 'Conditions',
           'CorrectnessConditions', 'Definiens', 'Directive', 'Environ', 'Equality',
           'Field-Segment', 'Format', 'Formats', 'Ident', 'Item', 'Iterative-Step', 'Label', 'Link',
           Loci´, 'LociEquality´, 'Locus´, 'NegatedAdjective´, 'Partial-Definiens´, 'Priority´,
           'Proposition', 'Provisional-Formulas', 'Redefine', 'Right-Circumflex-Symbol',
           Schematic-Variables, Scheme, Selector, SetMember, elSkippedProof, Symbol,
           'SymbolCount', 'Symbols', 'Substitution', 'Type-Specification', 'Type-List',
           'Variable', 'Variables', 'Vocabularies', 'Vocabulary');
  XMLAttrName: array [XMLAttrKind] of string = ('unknown', 'aid', 'argnr', 'articleid',
           'articleext', 'col', 'condition', 'constrnr', 'idnr', 'infinitive', 'kind', 'labelnr',
           'leftargnr', 'line', 'mizfiles', 'name', 'negated', 'nr', 'number', 'origin', 'posline',
           'poscol', 'priority', 'property', 'rightsymbolnr', 'schnr', 'serialnr', 'shape',
           spelling', 'symbolnr', 'value', 'varnr', 'varsort', 'x', 'x1', 'x2', 'y', 'y1', 'y2');
  implementation
  end.
```

Environment library

We have a library to handle accessing the Mizar mathematical library files. This is used in makeenv.dpr and using local prel/directories. This will execute InitLibrEnv (§482) and CheckCompatibility (§479). $\langle \text{ librenv.pas } 463 \rangle \equiv$ ⟨GNU License 4⟩ unit librenv; interface uses mobjects; ⟨Interface for MIZFILES library 464⟩ implementation uses $Q{Q&\$IFDEF\ WIN32Q}\ windows$, **Q{Q&\$***ENDIF***Q}***mizenv*, *pcmizver*, *mconsole*; ⟨Implementation for librenv.pas 467⟩ **begin** *InitLibrEnv*; *CheckCompatibility*; end. $\langle \text{Interface for MIZFILES library 464} \rangle \equiv$ const MML = `mml`; EnvMizFiles = `MIZFILES`; var MizPath, MizFiles: string; **function** LibraryPath(fName, fExt : string): string; **procedure** GetSortedNames(fParam : byte; var fList : MStringCollection); procedure GetNames(fParam : byte; var fList : StringColl); **procedure** ReadSortedNames(fName : string; var fList : MStringCollection); **procedure** ReadNames(fName : string; var fList : StringColl); See also section 465. This code is used in section 463. There are two public-facing classes. ⟨Interface for MIZFILES library 464⟩ +≡ **type** (Declare *FileDescr* data type 466) $\langle \text{ Declare } FileDescrCollection \text{ data type } 469 \rangle$ var LocFilesCollection: FileDescrCollection;

§466 Mizar Parser Environment library 141

```
File descriptors. We use file descriptors for things. These are just "a file name" and "a timestamp".
\langle \text{ Declare } FileDescr \text{ data type } 466 \rangle \equiv
  PFileDescr = \uparrow FileDescr;
  FileDescr = \mathbf{object} \ (MObject)
     nName: PString;
     Time: LongInt;
     constructor Init(fIdent : string; fTime : LongInt);
     destructor Done; virtual;
  end;
This code is used in section 465.
467.
       Constructor.
\langle \text{Implementation for librenv.pas } 467 \rangle \equiv
constructor FileDescr.Init(fIdent : string; fTime : LongInt);
  begin nName \leftarrow NewStr(fIdent); Time \leftarrow fTime;
  end;
See also sections 470, 479, and 482.
This code is used in section 463.
468. Destructor.
destructor FileDescr.Done:
  begin DisposeStr(nName);
  end;
469.
       Collection of file descriptions.
\langle \text{ Declare } FileDescrCollection \text{ data type } 469 \rangle \equiv
  PFileDescrCollection = \uparrow FileDescrCollection;
  FileDescrCollection = \mathbf{object} \ (MSortedCollection)
     function Compare(Key1, Key2 : Pointer): integer; virtual;
     procedure StoreFIL(fName : string);
     constructor LoadFIL(fName : string);
     procedure InsertTimes;
  end;
This code is used in section 465.
470. Comparing two entries in a file descriptor collection amounts to comparing the names for the file
descriptors.
\langle \text{Implementation for librenv.pas } 467 \rangle + \equiv
function FileDescrCollection.Compare(Key1, Key2 : Pointer): integer;
  begin if PFileDescr(Key1)\uparrow.nName\uparrow < PFileDescr(Key2)\uparrow.nName\uparrow then Compare \leftarrow -1
  else if PFileDescr(Key1)\uparrow.nName\uparrow = PFileDescr(Key2)\uparrow.nName\uparrow then Compare \leftarrow 0
     else Compare \leftarrow 1;
  end;
471. Inserting file times into the file descriptors relies upon mizenv.pas's GetFileTime (§37) function.
procedure FileDescrCollection.InsertTimes;
  var z: integer;
  begin for z \leftarrow 0 to Count - 1 do
     with PFileDescr(Items\uparrow[z])\uparrow do Time \leftarrow GetFileTime(nName\uparrow);
  end;
```

Mizar Parser §472

472. Constructor. This leverages a few primitive PASCAL functions: assign(file, name) assigns name to a file but does not open the file (it is still considered closed). Then reset(file) opens the file for reading.

Specifically, this will load a .fil file produced by Mizar. These contain 2N lines: a file path on line 2n-1, then a timestamp on line 2n for $n=1,\ldots,N$. This appears to be used for local prel/ files.

```
constructor FileDescrCollection.LoadFIL(fName : string);
  var FIL: text; lName: string; lTime: longint;
  begin Assign(FIL, fName); Reset(FIL); Init(0, 10);
  while \neg eof(FIL) do
    begin ReadLn(FIL, lName); ReadLn(FIL, lTime); Insert(new(PFileDescr, Init(lName, lTime)));
    end;
  close(FIL);
  end;
```

473. Repopulate .fil file. This will erase the file named fName, then assign to FIL that file, and rewrite(FIL) will open it for writing.

This will loop through every item in the caller's underlying collection, writing the file names and times to the .fil file.

```
procedure FileDescrCollection.StoreFIL(fName:string);

var FIL: text; i: integer;

begin EraseFile(fName); Assign(FIL,fName); Rewrite(FIL); InsertTimes;

for i \leftarrow 0 to Count - 1 do

with PFileDescr(Items\uparrow[i])\uparrow do

begin WriteLn(FIL,nName\uparrow); WriteLn(FIL,Time)

end;

Close(FIL);

end;
```

474. The library path tries to use the local version of a file, if it exists as tested with MFileExists (§35). Otherwise it looks at the Mizar MML version of a file, if it exists.

This returns the path to the file, as a string. If the file cannot be found either in the local prel directory or the MML prel directory, then it returns the empty string.

```
function LibraryPath(fName, fExt: string): string;
begin LibraryPath \leftarrow ```;
if MFileExists(`prel` + DirSeparator + fName + fExt) then
begin LocFilesCollection.Insert(New(PFileDescr, Init(`prel` + DirSeparator + fName + fExt, 0)));
LibraryPath \leftarrow `prel` + DirSeparator + fName + fExt; exit
end;
if MFileExists(MizFiles + `prel` + DirSeparator + fName[1] + DirSeparator + fName + fExt) then
LibraryPath \leftarrow MizFiles + `prel` + DirSeparator + fName[1] + DirSeparator + fName + fExt;
end;
```

 $\S475$ Mizar Parser ENVIRONMENT LIBRARY 143

```
This function actually is not used anywhere, so I am not sure why we have it.
procedure ReadSortedNames(fName : string; var fList : MStringCollection);
  var NamesFile: text;
  begin if fName[1] = 0 then
    begin Delete(fName, 1, 1); FileExam(fName); Assign(NamesFile, fName); Reset(NamesFile);
    fList.Init(100, 100);
    while \neg seekEof(NamesFile) do
      begin ReadLn(NamesFile, fName); fList.Insert(NewStr(fName));
      end;
    exit:
    end;
  fList.Init(2,10); fList.Insert(NewStr(fName));
  end;
476. Again, this function is not used anywhere, so I am not sure why we have it.
procedure ReadNames(fName : string; var fList : StringColl);
  var NamesFile: text;
  begin if fName[1] = 0 then
    begin Delete(fName, 1, 1); FileExam(fName); Assign(NamesFile, fName); Reset(NamesFile);
    fList.Init(10, 10);
    while \neg seekEof(NamesFile) do
      begin ReadLn(NamesFile, fName); fList.Insert(NewStr(fName));
    exit;
    end:
  fList.Init(2,10); fList.Insert(NewStr(fName));
  end;
      This function is used in lisvoc.dpr
477.
procedure GetSortedNames(fParam : byte; var fList : MStringCollection);
  var FileName: string; NamesFile: text; i: integer;
  begin if ParamCount < fParam then
    begin fList.Init(0,0); exit
    end;
  FileName \leftarrow ParamStr(fParam);
  if FileName[1] = `@` then
    begin Delete(FileName, 1, 1); FileExam(FileName); Assign(NamesFile, FileName);
    Reset(NamesFile); fList.Init(10, 10);
    while \neg seekEof(NamesFile) do
      begin ReadLn(NamesFile, FileName); fList.Insert(NewStr(TrimString(FileName)));
      end;
    exit;
    end;
  fList.Init(2,8); fList.Insert(NewStr(FileName));
  for i \leftarrow fParam + 1 to ParamCount do
    \textbf{begin} \ \textit{FileName} \leftarrow \textit{ParamStr}(i); \ \textit{fList.Insert}(\textit{NewStr}(\textit{FileName}));
    end;
  end;
```

§478

end;

```
Continuing with the "this is not used anywhere" theme, this function is not used anywhere.
procedure GetNames(fParam : byte; var fList : StringColl);
  var FileName: string; NamesFile: text; i: integer;
  \textbf{begin if} \ \textit{ParamCount} < \textit{fParam } \textbf{then}
    begin fList.Init(0,0); exit
    end;
  FileName \leftarrow ParamStr(fParam);
  if FileName[1] = `@` then
    begin Delete (FileName, 1, 1); FileExam (FileName); Assign (NamesFile, FileName);
    Reset(NamesFile); fList.Init(10, 10);
    while \neg seekEof(NamesFile) do
      begin ReadLn(NamesFile, FileName); fList.Insert(NewStr(TrimString(FileName)));
      end;
    exit;
    end;
  fList.Init(2,8); fList.Insert(NewStr(FileName));
  for i \leftarrow fParam + 1 to ParamCount do
    begin FileName \leftarrow ParamStr(i); fList.Insert(NewStr(FileName));
    end;
```

§479 Mizar Parser ENVIRONMENT LIBRARY 145

479. Check compatibility of Mizar with MML. We will load the mml.ini file for the MML version number, and we check it against the Mizar version. If they are not compatible, print a message to the screen, and halt as an error has occurred.

```
The mml.ini file looks something like:
```

```
[Mizar verifier]
MizarReleaseNbr=8
MizarVersionNbr=1
MizarVariantNbr=15
[MML]
NumberOfArticles=1493
MMLVersion=5.94
```

We will read line-by-line the mml.ini file to initialize several variables. This motivates the Try_read_ini_var macro.

```
define init\_val\_and\_end(\#) \equiv val(lLine, \#, lCode);
  define Try\_read\_ini\_var(\#) \equiv lPos \leftarrow Pos(\#, lLine);
         if lPos > 0 then
            begin delete(lLine, 1, lPos + 15); init\_val\_and\_end
\langle \text{Implementation for librenv.pas } 467 \rangle + \equiv
procedure CheckCompatibility;
  var lFile: text; lLine, lVer1, lVer2, l: string; lPos, lCode: integer;
     lMizarReleaseNbr, lMizarVersionNbr, lMizarVariantNbr: integer;
  begin (Open mml.ini file 480)
  lMizarReleaseNbr \leftarrow -1; \ lMizarVersionNbr \leftarrow -1; \ lMizarVariantNbr \leftarrow -1;
  while \neg seekEof(lFile) do
     begin ReadLn(lFile, lLine); Try_read_ini_var(`MizarReleaseNbr=')(lMizarReleaseNbr);
     Try_read_ini_var('MizarVersionNbr=')(lMizarVersionNbr);
     Try_read_ini_var( 'MizarVariantNbr=')(lMizarVariantNbr);
     end:
  close(lFile);
  (Assert MML version is compatible with Mizar version 481)
  end;
480. We open the $MIZFILES/mml.ini file for reading.
\langle \text{ Open mml.ini file } 480 \rangle \equiv
  FileExam(MizFiles + MML + `.ini`); Assign(lFile, MizFiles + MML + `.ini`); Reset(lFile);
This code is used in section 479.
```

481. We need to check the MML version is compatible with the Mizar version. If they are not compatible, raise an error, print a warning to the user, and halt here.

```
 \langle \text{Assert MML version is compatible with Mizar version 481} \rangle \equiv \\ \textbf{if } \neg ((l\textit{MizarReleaseNbr} = PC\textit{MizarReleaseNbr}) \land (l\textit{MizarVersionNbr} = PC\textit{MizarVersionNbr})) \textbf{ then} \\ \textbf{begin } Str(PC\textit{MizarReleaseNbr}, l); \ l\textit{Ver1} \leftarrow l; \ Str(PC\textit{MizarVersionNbr}, l); \ l\textit{Ver1} \leftarrow l\textit{Ver1} + `.` + l; \ Str(l\textit{MizarReleaseNbr}, l); \ l\textit{Ver2} \leftarrow l; \ Str(l\textit{MizarVersionNbr}, l); \ l\textit{Ver2} \leftarrow l\textit{Ver2} + `.` + l; \ Str(l\textit{MizarVersionNbr}, l); \ l\textit{Ver2} \leftarrow l\textit{Ver2} + `.` + l; \ D\textit{rawMessage}(\texttt{MizarLSystemLver.L}' + l\textit{Ver1} + \texttt{`LisLincompatibleLwithLtheLMMLLversionLimportedL(' + l\textit{Ver2} + `)`, \ `PleaseLcheckL' + \textit{MizFiles} + `mml.ini`); \ halt(1); \ end; \end{aligned}
```

This code is used in section 479.

146 ENVIRONMENT LIBRARY

Initialize library environment. This will try to initialize the *MizFiles* variable to be equal to the \$MIZFILES environment variable (if that environment variable exists) or the directory of the program being executed. This *MizFiles* will always end in a directory separator.

```
We also initalize MizFileName, EnvFileName, ArticleName, ArticleExt to be empty strings.
```

```
define append\_dir\_separator(\#) \equiv \mathbf{if} \ \#[length(\#)] \neq DirSeparator \ \mathbf{then} \ \# \leftarrow \# + DirSeparator;
\langle Implementation for librenv.pas 467 \rangle + \equiv
procedure InitLibrEnv;
  begin LocFilesCollection.Init(0,20); MizPath \leftarrow ExtractFileDir(ParamStr(0)); \langle Initialize Mizfiles 483 \rangle
   MizFileName \leftarrow ``; EnvFileName \leftarrow ``; ArticleName \leftarrow ``; ArticleExt \leftarrow ``;
  end;
```

483. Initalizing *Mizfiles* requires a bit of work. We first guess it based on environment variables. Then we need to ensure it is a directory path.

```
\langle \text{ Initialize } \textit{Mizfiles } 483 \rangle \equiv
   (Guess MizFiles from environment variables or executable path 484)
   if MizFiles ≠ ``then append_dir_separator(MizFiles);
   \textbf{if} \ \textit{MizFiles} = \texttt{``then} \ \textit{Mizfiles} \leftarrow \textit{DirSeparator};
This code is used in section 482.
```

484. When the \$MIZFILES environment variable is set, we just use it. When it is empty or missing, then we guess the path of the executable invoked.

```
\langle \text{Guess } MizFiles \text{ from environment variables or executable path } 484 \rangle \equiv
   MizFiles \leftarrow GetEnvStr(EnvMizFiles);
  if MizFiles =  then MizFiles \leftarrow MizPath;
This code is used in section 483.
```

 $\S485$ Mizar Parser INFO FILE HANDLING 147

File 12

Info file handling

```
485. I don't think this is actually used anywhere, but I am including it for completeness.
\langle \text{ info.pas } 485 \rangle \equiv
  (GNU License 4)
unit info;
  interface uses errhan;
  var InfoFile: text;
  procedure InfoChar(C:char);
  procedure InfoInt(I:integer);
  procedure InfoWord(C:char; I:integer);
  procedure InfoNewLine;
  procedure InfoString(S:string);
  procedure InfoPos(Pos:Position);
  {\bf procedure}\ {\it InfoCurPos};
  procedure OpenInfoFile;
  procedure CloseInfofile;
  implementation
  uses mizenv, mconsole;
  procedure InfoChar(C:char);
    begin write(InfoFile, C)
    end:
  procedure InfoInt(I:integer);
    begin write(InfoFile, I, ` \Box `)
    end;
  procedure InfoWord(C:char; I:integer);
    begin write(InfoFile, C, I, ` \Box `)
    end;
  procedure InfoNewLine;
    begin WriteLn(InfoFile)
    end;
  procedure InfoString(S:string);
    begin write(InfoFile, S)
    end;
  procedure InfoPos(Pos:Position);
    begin with Pos do write(InfoFile, Line, `\_\`, Col, `\_\`)
    end;
  procedure InfoCurPos;
    begin with CurPos do write(InfoFile, Line, `\_`, Col, `\_`)
    end;
```

148 INFO FILE HANDLING Mizar Parser $\S486$

```
486. There are a few helper functions which is more than "Write ⟨data type⟩ to info file".
var _InfoExitProc: pointer;
procedure InfoExitProc;
  begin CloseInfoFile; ExitProc ← _InfoExitProc;
  end;
procedure OpenInfoFile;
  begin Assign(InfoFile, MizFileName + `.inf`); Rewrite(InfoFile);
  WriteLn(InfoFile, `Mizared_article: "`, MizFileName, `"`); _InfoExitProc ← ExitProc;
  ExitProc ← @InfoExitProc;
  end;
procedure CloseInfofile;
  begin close(InfoFile)
  end;
  end.
```

 $\{487 \quad \text{Mizar Parser} \quad \text{XML PARSER} \quad 149$

File 13

XML Parser

487. The XML parser module is used for extracting information from XML files. It does not "validate" the XML (it's assumed to already be valid). The scanner chops up the input stream into tokens, then the parser makes this available as tokens for the user.

Just to review some terminology from XML:

- (1) A "tag" is a markup construct that begins with a "<" and ends with a ">". There are three types of tags:
 - (i) Start-tags: like "<foo>"
 - (ii) End-tags: like "</foo>"
 - (iii) Empty-element tags: like "
"
- (2) A "Element" is a logical document component that either (a) begins with a start-tag and ends with an end-tag, or (b) consists of an empty-element tag. The characters between the start-tag and end-tag (if any) are called its "Contents", and may contain markup including other elements which are called "Child Elements".
- (3) An "Attribute" is a markup construct consisting of a name-value pair which can exist in a start-tag or an empty-element tag. For example "" has two attributes: one named "src" whose value is "madonna.jpg", and the other named "alt" whose value is "Madonna".
- (4) XML documents may start with an "XML declaration" which looks something like (after some optional whitespace) "<?xml version="1.0" encoding="UTF-8"?>"

```
⟨ xml_parser.pas 487⟩ ≡
⟨ GNU License 4⟩
unit xml_parser;
interface uses mobjects, errhan;
⟨ Constants for xml_parser.pas 488⟩
⟨ Type declarations for xml_parser.pas 489⟩
procedure XMLASSERT(aCond: boolean);
procedure UnexpectedXMLElem(const aElem: string;
    aErr: integer);
implementation
    mdebug;
uses info;
    end_mdebug;
⟨ Implementation of XML Parser 491⟩
end .
```

150 XML PARSER Mizar Parser §488

488. Constant parameters. We have a few constant parameters for the error codes.

```
 \begin{array}{l} \langle \, \text{Constants for xml\_parser.pas 488} \, \rangle \equiv \\ \mathbf{const} \, \, \mathit{InOutFileBuffSize} = \$4000; \\ \{ \, \text{for xml attribute tables} \, \} \\ \mathbf{const} \, \, \mathit{errElRedundant} = 7500; \quad \{ \, \text{End of element expected, but child element found} \, \} \\ \mathbf{const} \, \, \mathit{errElMissing} = 7501; \quad \{ \, \text{Child element expected, but end of element found} \, \} \\ \mathbf{const} \, \, \mathit{errMissingXMLAttribute} = 7502; \quad \{ \, \text{Required XML attribute not found} \, \} \\ \mathbf{const} \, \, \mathit{errWrongXMLElement} = 7503; \quad \{ \, \text{Different XML element expected} \, \} \\ \mathbf{const} \, \, \mathit{errBadXMLToken} = 7506; \quad \{ \, \text{Unexpected XML token} \, \} \\ \text{This code is used in section 487.} \end{array}
```

489. Public type declarations. We will defer the "PASCAL classes" until we start implementing them. Right now, we have syntactic classes for the tokens. Specifically we have the start of an XML declaration "<?", the end of an XML declaration "?>", the start of a character data section "<!", the start and end of tags, quotation marks, equalities, entities, identifiers, and end of text.

```
\langle \text{Type declarations for xml_parser.pas } 489 \rangle \equiv
type XMLTokenKind = (Err, {an error symbol})
  BI, \{ \langle ? \rangle \}
  EI, \{?>\}
  DT, \{ <! \}
  LT, \{ < \}
  GT, \{>\}
  ET,
         { </ }
  EE,
        {/>}
  QT, {"}
  EQ, \{=\}
  EN, { Entity }
  ID, { Identifier, Name }
  EOTX); { End of text }
  TokensSet = \mathbf{set} \ \mathbf{of} \ XMLTokenKind;
  (Declare XML Scanner Object type 494)
  TElementState = (eStart, eEnd); { high-level parser states, see procedure NextElementState}
  (Declare XML Attribute Object 490)
  ⟨ Declare XML Parser object 502⟩
This code is used in section 487.
```

490. XML Attribute Object. An XML attribute contains the attribute name and its value. We can represent it as "just" an *MStrObj* (§191) with an additional "value" field.

This code is used in section 489.

 $\S491$ Mizar Parser XML PARSER 151

```
Constructor. This uses the MStrObj. Init constructor to initialize the name, then it sets the value.
\langle Implementation of XML Parser 491\rangle \equiv
constructor XMLAttrObj.Init(const aName, aValue: string);
  begin inherited Init(aName); nValue \leftarrow aValue;
  end:
See also sections 492, 493, 495, 497, 498, 501, 503, 508, and 510.
This code is used in section 487.
      Assertion. We have a helper function for asserting things about XML. This is just a wrapper around
MizAssert (§113).
\langle Implementation of XML Parser 491\rangle + \equiv
procedure XMLASSERT (aCond: boolean);
  begin MizAssert(errWrongXMLElement, aCond);
  end;
493.
       Unexpected XML Element. Another helper function for checking XML parsing.
\langle Implementation of XML Parser 491\rangle + \equiv
procedure UnexpectedXMLElem(const aElem: string;
  aErr: integer);
    mdebug;
  var lEl: string;
    end_mdebug;
  mdebug ; InfoNewLine; end_mdebug;
  RunTimeError(aErr);
  end;
      XML Scanner Object. The scanner produces a stream of tokens, which is then consumed by the
XML parser. Hence, besides the constructor and destructor, there is only one public facing method: get the
next token.
\langle Declare XML Scanner Object type 494\rangle \equiv
  XMLScannObj = \mathbf{object} (MObject)
    nSourceFile: text;
    nSourceFileBuff: pointer;
    nCurTokenKind: XMLTokenKind;
    nSpelling: string;
    nPos: Position;
    nCurCol: integer;
    nLine: string;
    constructor InitScanning(const aFileName: string);
    destructor Done; virtual;
    procedure GetToken; private
    procedure GetAttrValue;
  end;
This code is used in section 489.
```

152 XML PARSER Mizar Parser $\S495$

Constructor. We open the file (doing all the boilerplate file IO stuff), then initialize the fields of the scanner to prepare to read the first line from the file. $\langle \text{Implementation of XML Parser 491} \rangle + \equiv$ **constructor** XMLScannObj.InitScanning(**const** aFileName: string); **begin** inherited Init; (Prepare to read in the contents of XML file 496); $nSpelling \leftarrow \ \ \ \ \ \ nLine \leftarrow \ \ \ \ \ nCurCol \leftarrow 0; \ nPos.Line \leftarrow 0; \ nPos.Col \leftarrow 0;$ GetToken;end; 496. This prepares to read in from an XML file, setting up a text buffer, and opening the file in "read mode". \langle Prepare to read in the contents of XML file 496 $\rangle \equiv$ Assign(nSourceFile, aFileName); GetMem(nSourceFileBuff, InOutFileBuffSize); $SetTextBuf(nSourceFile, nSourceFileBuff\uparrow, InOutFileBuffSize); Reset(nSourceFile)$ { open for reading } This code is used in section 495. **Destructor.** We need to close the XML file, as well as free up the input buffer. \langle Implementation of XML Parser 491 $\rangle + \equiv$ **destructor** *XMLScannObj.Done*; **begin** close(nSourceFile); FreeMem(nSourceFileBuff, InOutFileBuffSize); $nLine \leftarrow ``; nSpelling \leftarrow ``;$ inherited Done; end; Getting the token. The scanner produces tokens on demand. They are assembled into a tree data 498. structure by the parser. This method may look a bit foreign, since it's a procedure and not a function. The current token is stored in several fields in the scanner. The token's lexeme is stored into the nSpelling field. **define** $update_lexeme \equiv nSpelling \leftarrow Copy(nLine, nPos.Col, nCurCol - nPos.Col)$ ⟨Implementation of XML Parser 491⟩ +≡ procedure XMLScannObj.GetToken; const CharKind: array [chr(0) ... chr(255)] of - . / 0 1 2 3 4 5 6 7 8 9 : ; } { 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 { abcdefghijklmnopqrstuvwxyz} **begin** (Skip whitespace for XML parser 499); $nPos.Col \leftarrow nCurCol;$ (Get token kind based off of leading character 500);

while $(nCurCol < length(nLine)) \land (nLine[nCurCol] \in [`__`, `_])$ do inc(nCurCol);

 $update_lexeme$;

end:

 $\{499 \quad \text{Mizar Parser}$ XML PARSER 153

499. If we're done in the file, then we've arrived at the "end-of-file" — i.e., eof(nSourceFile) is true. In this case, the token returned should be an EOTX (end of text). We also end the function here.

On the other hand, if there is still more left in the file, we should read in a line, increment the line number, reset the column to 1, and skip over any whitespace (specifically, "SP" are skipped over — tabs or newlines are not skipped).

```
 \langle \text{Skip whitespace for XML parser } 499 \rangle \equiv \\ \textbf{while } nCurCol = length(nLine) \textbf{ do} \\ \textbf{begin if } eof(nSourceFile) \textbf{ then} \\ \textbf{begin } nCurTokenKind \leftarrow EOTX; \ nSpelling \leftarrow \texttt{``}; \ exit \textbf{ end}; \\ ReadLn(nSourceFile, nLine); \ inc(nPos.Line); \ nLine \leftarrow nLine + \texttt{`}_{\sqcup}\texttt{`}; \ nCurCol \leftarrow 1; \\ \textbf{while } (nCurCol < length(nLine)) \land (nLine[nCurCol] = \texttt{`}_{\sqcup}\texttt{`}) \textbf{ do } inc(nCurCol); \\ \textbf{end} \\ \end{aligned}
```

500. There are several situations when determining tokens. We will often want to keep accumulating alphanumeric characters, so we describe this in the "keep eating alphadigits" macro.

When we encounter a "<" character, this could begin or end a tag, or it could be something special if the next character is "?" or "!". We determine the type in the "get tag kind" macro.

```
define keep\_eating\_alphadigits \equiv
           begin nCurTokenKind \leftarrow ID;
           repeat inc(nCurCol)
           until CharKind[nLine[nCurCol]] = 0;
  define qet\_taq\_kind \equiv inc(nCurCol);
         case nLine[nCurCol] of
         '/: begin nCurTokenKind \leftarrow ET; inc(nCurCol); end;
         ??: begin nCurTokenKind \leftarrow BI; inc(nCurCol); end;
         :: begin nCurTokenKind \leftarrow DT; inc(nCurCol); end;
         othercases nCurTokenKind \leftarrow LT;
         endcases
  define keep\_getting\_until\_end\_of\_tag(\#) \equiv \mathbf{begin} \ inc(nCurCol);
         if nLine[nCurCol] = \rightarrow then
           begin nCurTokenKind \leftarrow \#; inc(nCurCol); end
         else nCurTokenKind \leftarrow Err;
         end:
\langle Get token kind based off of leading character 500\rangle \equiv
  case nLine[nCurCol] of
  'a'..'z', 'A'...'Z', '0'...'9', '_', '-', '&': keep_eating_alphadigits;
  ": begin nCurTokenKind \leftarrow QT; inc(nCurCol) end;
  '=': begin nCurTokenKind \leftarrow EQ; inc(nCurCol) end;
  '<': begin qet_taq_kind; end;
  \rightarrow: begin nCurTokenKind \leftarrow GT; inc(nCurCol) end;
  '/': keep_getting_until_end_of_tag(EE);
  : keep_qetting_until_end_of_tag(EI);
  othercases begin nCurTokenKind \leftarrow Err; inc(nCurCol) end;
  endcases
```

This code is used in section 498.

This code is used in section 498.

154 XML PARSER Mizar Parser $\S 501$

501. Scanners can obtain attribute values as tokens. This is used by the XML parser (§§506, 508). I think one possible source of bugs is that this does not handle escaped quotes (e.g., "\"" is traditionally parsed as a quotation mark character).

```
This will not include the delimiting quotation marks, and it will also skip all whitespace after the attribute.
  define skip\_to\_quotes \equiv \mathbf{while} \ (nCurCol < length(nLine)) \land (nLine[nCurCol] \neq `"`) do inc(nCurCol)
  define is\_space \equiv (nCurCol < length(nLine)) \land (nLine[nCurCol] \in [`\_', ` `])
  define skip\_spaces \equiv while is\_space do inc(nCurCol)
\langle Implementation of XML Parser 491\rangle + \equiv
procedure XMLScannObj.GetAttrValue;
  var lCol: integer;
  begin lCol \leftarrow nCurCol; skip\_to\_quotes;
  nSpelling \leftarrow Copy(nLine, lCol, nCurCol - lCol); { save the lexeme }
  if nLine[nCurCol] = ``` then inc(nCurCol);
  skip\_spaces;
  end;
       XML Parser. We recall (\S489) the type for element states (it's an enumerated type with two values,
eStart and eEnd).
\langle \text{ Declare XML Parser object 502} \rangle \equiv
  XMLParserObj = \mathbf{object} (XMLScannObj)
    nElName: string; { name of the current element }
    nState: TElementState;
    nAttrVals: MSortedStrList;
    constructor InitParsing(const aFileName: string);
    destructor Done; virtual;
    procedure ErrorRecovery(aErr: integer; aSym: TokensSet);
    procedure NextTag; virtual;
    procedure NextElementState; virtual;
    procedure AcceptEndState; virtual;
    procedure AcceptStartState; virtual;
    procedure OpenStartTag; virtual;
    procedure CloseStartTag; virtual;
    \textbf{procedure} \ \textit{CloseEmptyElementTag}; \ \textit{virtual};
    procedure ProcessEndTag; virtual;
    procedure ProcessAttributeName; virtual;
    procedure ProcessAttributeValue; virtual;
    procedure SetAttributeValue(const aVal: string);
  end:
This code is used in section 489.
```

 $\S503$ Mizar Parser XML PARSER 155

503. Constructor. The parser expects an XML file to start with "<?xml ...?>" (everything after the "xml" is ignored). If this is not the first non-whitespace entry, an error will be raised. The constructor will then skip all other "<?...?>" entities.

define $skip_xml_prolog \equiv \mathbf{while} \ (nCurTokenKind \neq EOTX) \land (nCurTokenKind \neq EI) \ \mathbf{do} \ GetToken;$

```
if nCurTokenKind = EI then GetToken define skip\_all\_other\_ids \equiv while nCurTokenKind = BI do begin GetToken; while (nCurTokenKind \neq EOTX) \land (nCurTokenKind \neq EI) do GetToken; if nCurTokenKind \neq EOTX) \land (nCurTokenKind \neq EI) do GetToken; if nCurTokenKind = EI then GetToken; end

(Implementation of XML Parser 491) +\equiv constructor XMLParserObj.InitParsing (const aFileName: string); begin inheritedInitScanning (aFileName); nElName \leftarrow ``; nAttrVals.Init(0); if nCurTokenKind = BI then begin GetToken; if (nCurTokenKind = ID) \land (nSpelling = `xml`) then GetToken else ErrorRecovery (10, [EI, LT]); skip\_xml\_prolog; skip\_all\_other\_ids; { skip all other initial processing instructions } end; end;
```

504. Destructor. We will set the element name to the empty string, and invoke the destructor for the attribute values.

```
destructor XMLParserObj.Done;
begin inherited Done; nAttrVals.Done; nElName ← ´´;
end;
```

505. Error recovery. We just raise a runtime error. In fact, this is often used in situations like:

```
if nCurTokenKind = ID then { success } else ErrorRecovery(5, [LT, ET]);
```

Consequently, it is probably more idiomatic to introduce a macro $xml_match(tokenKind)(aErr, aSym)$ to assert the match and raise an error for mismatch. Unfortunately, WEB macros allow for only one argument, so we need two macros.

```
define report\_mismatch(\#) \equiv ErrorRecovery(\#)
define xml\_match(\#) \equiv \mathbf{if} \ nCurTokenKind \neq \# \mathbf{then} \ report\_mismatch
{ ErrorRecovery is no longer allowed for XML, bad XML is just RTE}
procedure XMLParserObj.ErrorRecovery(aErr:integer; aSym:TokensSet);
begin Mizassert(errBadXMLToken,false);
end;
```

156 XML PARSER Mizar Parser $\S506$

```
The parser will the consume the next tag or element. It's useful to recall the token kinds (§489).
  Curiously, the attributes are skipped during this parsing function.
  This will be using the inherited procedure GetToken (§498).
    { Parses next part of XML, used for skipping some part of XML }
    { setting the nState to eStart or eEnd.}
    \{ nElName \text{ is set properly } \}
    \{ nAttrVals \text{ are omitted (skiped).} \}
procedure XMLParserObj.NextTag;
  begin case nCurTokenKind of
  EOTX: nState \leftarrow eEnd;  { sometimes we need this }
  LT: begin nState \leftarrow eStart; GetToken; xml\_match(ID)(6, [LT, ET]); OpenStartTaq; GetToken;
    \langle \text{ Get contents of XML start tag 507} \rangle;
    end;
  EE: begin nState \leftarrow eEnd; GetToken; end;
  ET: begin nState \leftarrow eEnd; GetToken; xml\_match(ID)(8, [LT, ET]); OpenStartTag; GetToken;
    xml_{-}match(GT)(7, [LT, ET]); GetToken
  othercases ErrorRecovery(9, [LT, ET]);
  endcases;
  end;
507. When getting the contents of an XML start tag (or possibly an element), we keep going until we get
to either "\>" (for an element) or ">" (for a tag). This will be using the inherited procedure GetToken
(\S498).
  define qet\_attribute \equiv \mathbf{begin} \ GetToken; \ xml\_match(EQ)(4, [ID, GT, LT, ET]); \ GetToken;
         xml\_match(QT)(3, [ID, GT, LT, ET]); GetAttrValue; GetToken;
         end
\langle \text{ Get contents of XML start tag 507} \rangle \equiv
  repeat case nCurTokenKind of
    GT: begin GetToken; break end;
    EE: begin break end;
    ID: qet_attribute;
    othercases begin ErrorRecovery(5, [GT, LT, ET]); break end;
    endcases;
  until nCurTokenKind = EOTX
This code is used in section 506.
```

 $\{508 \quad \text{Mizar Parser} \quad \text{XML PARSER} \quad 157$

508. For Mizar, *everything* will be encoded as an element or an attribute on an element. So we do not really need to consider the case where we would encounter text in the body of an element.

```
⟨ Implementation of XML Parser 491⟩ +≡
{ Parses next part of XML, setting the nState to eStart or eEnd. If nState = eStart, then nElName, nAttrVals are set properly. It is possible to go from nState = eStart to nState = eStart (when the element is non empty), and similarly from eEnd to eEnd. }

procedure XMLParserObj. NextElementState;

begin case nCurTokenKind of

EOTX: nState \leftarrow eEnd; {sometimes we need this}

LT: ⟨Parse start of XML tag 509⟩;

EE: begin nState \leftarrow eEnd; GetToken; end;

ET: begin nState \leftarrow eEnd; GetToken; xml\_match(ID)(8, [LT, ET]); ProcessEndTag; GetToken; xml\_match(GT)(7, [LT, ET]); GetToken; end;

othercases ErrorRecovery(9, [LT, ET]); endcases; end;
```

509. We start parsing a start-tag because we have encountered an LT token. So at this point, the next token should be an identifier of some kind. A start-tag may actually be an empty-element tag, so we need to look out for the EE token kind.

Note: the XML parser does not handle comments, otherwise we would need to consider that situation here.

```
define end\_start\_tag \equiv \mathbf{begin} GetToken; CloseStartTag; break end
  define end_empty\_tag \equiv \mathbf{begin} CloseEmptyElementTag; break end
\langle \text{ Parse start of XML tag 509} \rangle \equiv
  begin nState \leftarrow eStart; GetToken; xml\_match(ID)(6, [LT, ET]); OpenStartTaq;
       { Start-Tag or Empty-Element-Tag Name = nSpelling }
  GetToken:
  repeat case nCurTokenKind of
    GT: end\_start\_tag;  { End of a Start-Tag }
    EE: end_empty_tag; { End of a Empty-Element-Tag }
    ID: \mathbf{begin} \ ProcessAttributeName; \ GetToken; \ xml\_match(EQ)(4, [ID, GT, LT, ET]); \ GetToken;
       xml\_match(QT)(3,[ID,GT,LT,ET]); GetAttrValue; ProcessAttributeValue; GetToken;
    othercases begin ErrorRecovery(5, [GT, LT, ET]); break end;
    endcases;
  until nCurTokenKind = EOTX;
  end
This code is used in section 508.
510. We will want assertions reflecting the parser is in a "start" state or an "end" state.
\langle \text{Implementation of XML Parser 491} \rangle + \equiv
procedure XMLParserObj.AcceptEndState;
  begin NextElementState; MizAssert(errElRedundant, nState = eEnd);
  end;
procedure XMLParserObj.AcceptStartState;
  begin NextElementState; MizAssert(errElMissing, nState = eStart);
  end:
```

158 XML PARSER Mizar Parser $\S 511$

```
procedure XMLParserObj.OpenStartTag;
  begin nElName \leftarrow nSpelling; nAttrVals.FreeAll;
  end;
      We have a few procedures which are, well, empty. I am not sure why we have them. Regardless, here
they are!
procedure XMLParserObj.CloseStartTag;
  begin end;
procedure XMLParserObj.CloseEmptyElementTag;
  begin end;
procedure XMLParserObj.ProcessEndTag;
  begin end;
513. We have a list of attributes. When the parser ProcessAttributeName, it will merely push a new
XMLAttrPtr to the list with the given name. Then ProcessAttributeValue will associate to it the value
which has been parsed. We can, of course, manually set the value for an attribute using SetAttribute Value.
procedure XMLParserObj.ProcessAttributeName;
  begin nAttrVals.Insert(new(XMLAttrPtr, Init(nSpelling, ``)));
  end;
procedure XMLParserObj.ProcessAttributeValue;
  begin SetAttributeValue(nSpelling);
  end;
procedure XMLParserObj.SetAttributeValue(const aVal: string);
  begin with nAttrVals do XMLAttrPtr(Items\uparrow[Count-1])\uparrow.nValue \leftarrow aVal;
```

end;

 $\S514$ Mizar Parser I/O WITH XML 159

File 14

This code is used in section 514.

I/O with XML

514. We will want to print some XML to a buffer or stream.

Note that XML seems to be frozen at version 1.0 (first published in 1998, last revised in its fifth edition released November 26, 2008).

```
\langle \text{xml_inout.pas } 514 \rangle \equiv
  ⟨GNU License 4⟩
unit xml_inout;
  interface
  uses errhan, mobjects, xml_parser;
  \langle Type declarations for XML I/O 515\rangle
  function QuoteStrForXML(const aStr: string): string;
  function XMLToStr(const aXMLStr: string): string;
  function QuoteXMLAttr(aStr: string): string;
  const gXMLHeader = `<?xml_version="1.0"?>` + #10;
  implementation
  uses SysUtils, mizenv, pcmizver, librenv, xml_dict
  mdebug , info end_mdebug;
\langle \text{Implementation for I/O of XML 516} \rangle
end .
       There are only 4 types of streams we care about: Streams, Text Streams, XML Input Streams, and
515.
XML Output Streams.
\langle \text{Type declarations for XML I/O 515} \rangle \equiv
  ⟨ Public interface for XML Input Stream 528⟩;
  ⟨Public declaration for Stream Object 520⟩;
  (Public declaration for Text Stream Object 524);
  ⟨ Public declaration for XML Output Stream 533⟩;
```

160 I/O WITH XML Mizar Parser $\S516$

516. Escape for quote string. We want to allow only alphanumerics [a-zA-Z0-9] as well as dashes ("-"), spaces (""), commas (",") periods ("."), apostrophes ("'"), forward slashes ("/"), underscores ("-"), brackets ("[" and "]"), exclamation points ("!"), semicolons and colons (";" and ":"), and equal signs ("="). Everything else we transform into an XML entity of the form "&xx" where x is a hexadecimal digit.

517. This appears to "undo" the previous function, transforming XML entities of the form "&xx" into characters.

```
function XMLToStr(const aXMLStr: string): string;

var i, h: integer; lHexNr: string;

begin Result \leftarrow aXMLStr;

for i \leftarrow length(Result) - 5 downto 1 do

begin \langle Transform XML entity into character, if encountering an XML entity at i 518\rangle;

end;

Result \leftarrow Trim(Result);

end;
```

518. Transforming an XML entity into a character. This specifically checks for *hexadecimal* entities of the form "&#xXX" for some hexadecimal digits X. Note we must prepend "ox" to a numeric string for PASCAL to parse it as hexadecimal.

Since PASCAL does not have shortcircuiting Boolean operations, we need to make this a nested if statement.

```
⟨ Transform XML entity into character, if encountering an XML entity at i 518⟩ ≡ if (Result[i] = \mathring{x}) \land (length(Result) \ge i + 5) then begin if (Result[i+1] = \mathring{x}) \land (Result[i+2] = \mathring{x}) then begin lHexNr \leftarrow Result[i+3] + Result[i+4]; h \leftarrow StrToInt(\mathring{x} + lHexNr); Delete(Result, i, 5); Result[i] \leftarrow chr(h); end; end
```

This code is used in section 517.

 $\S519$ Mizar Parser I/O WITH XML 161

519. We can quote an XML attribute, escaping quotes, ampersands, and angled brackets. For non-ASCII characters, we escape it to a hexadecimal XML entity.

```
\langle Implementation for I/O of XML 516\rangle +=
function QuoteXMLAttr(aStr:string): string;
  var i: integer;
  begin Result \leftarrow ::
  for i \leftarrow 1 to length(aStr) do
     case aStr[i] of
     ": Result \leftarrow Result + `\"`;
     '&': Result \leftarrow Result + \text{`\&'};
     <: Result \leftarrow Result + `&lt;`;
     \rightarrow: Result \leftarrow Result + '>';
     othercases if integer(aStr[i]) > 127 then
          Result \leftarrow Result + \text{`\&\#x'} + IntToHex(Ord(aStr[i]), 2) + \text{`;'}
       else Result \leftarrow Result + aStr[i];
     endcases:
  end;
       Stream object class. A stream consists of a file, a character buffer, as well as integers tracking the
size of the buffer and (I think) the position in the buffer. This is the parent class to XML output buffers.
\langle \text{Public declaration for Stream Object 520} \rangle \equiv
  StreamObj = \mathbf{object} \ (MObject)
     nFile: File;
     fFileBuff: \uparrow BuffChar;
    fBuffCount, fBuffInd: longint;
     constructor InitFile(const AFileName: string);
     procedure Error(Code, Info: integer); virtual;
     destructor Done; virtual;
  end
This code is used in section 515.
521. We will have a wrapper function for conveniently reporting errors.
\langle \text{Implementation for I/O of XML 516} \rangle + \equiv
procedure StreamObj.Error(Code, Info: integer);
  begin RunError(2000 + Code);
  end;
       Constructor. We begin by Assign-ing a name to a file, allocating a file buffer, then initializing the
buffer size to zero, and the buffer position to zero. (The buffer position fBuffInd is needed only when writing
to an output XML stream.)
constructor StreamObj.InitFile(const AFileName: string);
  begin Assign(nFile, AFileName); new(fFileBuff); fBuffCount <math>\leftarrow 0; fBuffInd \leftarrow 0;
  end;
      Destructor. We close the file, and free up the file buffer.
destructor StreamObj.Done;
  begin Close(nFile); dispose(fFileBuff);
  end:
```

162 I/O WITH XML Mizar Parser $\S524$

```
524.
       Text Stream Object. A text stream is very similar to a Stream Object, except it is specifically for
text.
\langle Public declaration for Text Stream Object 524\rangle \equiv
  TXTStreamObj = \mathbf{object} \ (MObject)
    nFile: text;
    nFileBuff: pointer;
    constructor InitFile(const AFileName: string);
    procedure Error(Code, Info: integer); virtual;
    destructor Done; virtual;
  end
This code is used in section 515.
525. We have the convenience function for reporting errors.
\langle \text{Implementation for I/O of XML 516} \rangle + \equiv
procedure TXTStreamObj.Error(Code, Info: integer);
  begin RunError(2000 + Code);
  end;
526.
       Constructor. Assign a name to the file, allocate an input buffer, then initialize the buffer.
constructor TXTStreamObj.InitFile(const AFileName: string);
  begin Assign(nFile, AFileName); GetMem(nFileBuff, InOutFileBuffSize);
  SetTextBuf(nFile, nFileBuff\uparrow, InOutFileBuffSize);
  end;
527. Destructor. Simply free the underlying file buffer.
destructor TXTStreamObj.Done;
  begin FreeMem(nFileBuff, InOutFileBuffSize);
  end;
      XML Input Streams. An input stream reads an XML file and produces an abstract syntax tree
for its contents. This extends this XML parser class (§502). It may be tempting to draw similarities with,
e.g., the StAX library (in Java), but the truth is there's only finitely many ways to parse XML, and some
ways are just more natural.
\langle \text{ Public interface for XML Input Stream 528} \rangle \equiv
  XMLInStreamPtr = \uparrow XMLInStreamObj;
  XMLInStreamObj = \mathbf{object} \ (XMLParserObj)
    constructor OpenFile(const AFileName: string);
    function GetOptAttr (const aAttrName: string; var aVal: string): boolean;
    function GetAttr(const aAttrName: string): string;
    function GetIntAttr(const aAttrName: string): integer;
  end
This code is used in section 515.
529. Constructor. The non-debugging code just invokes the XML Parser's constructor (§503).
\langle \text{Implementation for I/O of XML 516} \rangle + \equiv
constructor XMLInStreamObj.OpenFile(const AFileName: string);
  mdebug; write(InfoFile, AFileName); end_mdebug;
  InitParsing(AFileName);
  mdebug ; WriteLn(InfoFile, `\_\reset'); end_mdebug;
```

end;

 $\S530$ Mizar Parser I/O WITH XML 163

530. We use the inherited *XMLParserObj*'s *nAttrVals*: *MSortedStrList* to track the XML attributes. If *aAttrName* is stored there, this will mutate *aVal* to store the associated value and the function will return *true*. Otherwise, this will return *false*.

This is useful for getting the value of an optional XML attribute.

```
{ get string denoted by optional XML attribute aAttrName }
```

```
function XMLInStreamObj.GetOptAttr (const aAttrName: string; var aVal: string): boolean; var lAtt: XMLAttrPtr; begin lAtt \leftarrow XMLAttrPtr (nAttrVals.ObjectOf (aAttrName)); if lAtt \neq nil then begin aVal \leftarrow lAtt \uparrow .nValue; GetOptAttr \leftarrow true; exit; end; GetOptAttr \leftarrow false; end;
```

531. When we know an XML attribute is *required*, we can just get the associated value directly (and raise an error if it is missing).

```
{ get string denoted by required XML attribute aAttrName } function XMLInStreamObj.GetAttr(\mathbf{const}\ aAttrName:\ string):\ string; var lAtt:\ XMLAttrPtr; begin lAtt \leftarrow XMLAttrPtr(nAttrVals.ObjectOf(aAttrName)); if Latt \neq \mathbf{nil}\ \mathbf{then}
```

```
begin GetAttr \leftarrow lAtt\uparrow.nValue; exit; end; MizAssert(errMissingXMLAttribute, false); end:
```

532. When the required attribute has an integer value, we should return the integer-value of it. Does this ever happen? Yes! For example, when writing an article named article.miz, then we run the verifier on it, we shall obtain article.xml which will contain tags of the form "<Adjective nr="5">"."

```
\{\,{\rm get}\ {\rm integer}\ {\rm denoted}\ {\rm by}\ {\rm required}\ {\rm XML}\ {\rm attribute}\ {\rm aAttrName}\,\}
```

```
function XMLInStreamObj.GetIntAttr(\mathbf{const}\ aAttrName:\ string):\ integer; var lInt, ec:\ integer; begin val(GetAttr(aAttrName), lInt, ec);\ GetIntAttr \leftarrow lInt; end;
```

164 I/O WITH XML Mizar Parser §533

533. XML Output Streams. We will want to write data to an XML file. This gives us an abstraction for doing so.

```
\langle Public declaration for XML Output Stream 533\rangle \equiv
  XMLOutStreamPtr = \uparrow XMLOutStreamObj;
  XMLOutStreamObj = \mathbf{object} (StreamObj)
    nIndent: integer; { indenting }
    constructor OpenFile(const AFileName: string);
    constructor OpenFileWithXSL(const AFileName: string);
    destructor EraseFile;
    procedure OutChar(AChar : char);
    procedure OutNewLine;
    procedure OutString(const AString: string);
    procedure OutIndent;
    procedure Out_XElStart(const fEl: string);
    procedure Out_XAttrEnd;
    procedure Out_XElStart0(const fEl: string);
    procedure Out_XElEnd0;
    procedure Out_XEl1(const fEl: string);
    procedure Out_XElEnd(const fEl: string);
    procedure Out_XAttr(const fAt, fVal: string);
    procedure Out_XIntAttr(const fAt: string;
      fVal: integer);
    procedure Out_PosAsAttrs(const fPos: Position);
    procedure Out_XElWithPos(const fEl: string;
      const fPos: Position);
    procedure Out_XQuotedAttr(const fAt, fVal: string);
    destructor Done; virtual;
  end
This code is used in section 515.
534. Constructor. We initialize a file, open it for writing, set the initial indentation amount to zero, and
then print the XML header declaration.
\langle \text{Implementation for I/O of XML 516} \rangle + \equiv
constructor XMLOutStreamObj.OpenFile(const AFileName: string);
  mdebug write(InfoFile, MizFileName + `.` + copy(AFileName, length(AFilename) - 2, 3));
      end_mdebug
  InitFile(AFileName); Rewrite(nFile, 1);
  mdebug WriteLn(InfoFile, '__rewritten'); end_mdebug
  nIndent \leftarrow 0; OutString(qXMLHeader);
  end:
      Constructor. Since XML supports custom style declarations (think of XSLT), we can also support
writing an XML file which uses them. This specifically needs to adjust the XML declaration.
    { add the stylesheet processing info }
constructor XMLOutStreamObj.OpenFileWithXSL(const AFileName: string);
  begin OpenFile(AFileName);
  OutString(```?xml-stylesheet_itype="text/xml", href="file://`+MizFiles+`miz.xml"?>``+#10):
  end;
```

 $\S536$ Mizar Parser I/O WITH XML 165

```
Destructor. We need to flush the buffer to the file before freeing up the buffer.
destructor XMLOutStreamObj.Done;
  begin if (fBuffInd > 0) \land (fBuffInd < InOutFileBuffSize) then
    BlockWrite(nFile, fFileBuff \uparrow, fBuffInd, fBuffCount);
  inherited Done;
  end;
537. Destructor. Some times we want to further erase the output file (which seems, at first glance, like
a really bad idea...).
destructor XMLOutStreamObj.EraseFile;
  begin Done; Erase(nFile);
  end:
538.
       Writing a character to the buffer. When the buffer is full, we flush it.
procedure XMLOutStreamObj.OutChar(aChar: char);
  begin fFileBuffInd \uparrow [fBuffInd] \leftarrow AnsiChar(aChar); inc(fBuffInd); \langle Flush XML output buffer, if full 539 <math>\rangle;
  end;
       The XML output buffer is full when the logical size (fBuffInd) is equal to the InOutFileBuffSize.
When this happens, we should write everything to the file, then reset the logical size parameter to zero.
\langle Flush XML output buffer, if full 539 \rangle \equiv
  if fBuffInd = InOutFileBuffSize then
    begin BlockWrite(nFile, fFileBuff \uparrow, InOutFileBuffSize, fBuffCount); fBuffInd <math>\leftarrow 0;
    end
This code is used in section 538.
540. Print a newline ("\n") to the XML output stream.
\langle \text{Implementation for I/O of XML 516} \rangle + \equiv
procedure XMLOutStreamObj.OutNewLine;
  begin OutChar(#10);
  end:
541. Printing a string to the output buffer.
procedure XMLOutStreamObj.OutString(const aString: string);
  var i: integer;
  begin for i \leftarrow 1 to length(aString) do OutChar(aString[i]);
  end;
542. Printing nIndent spaces ("_{\perp}") to the output buffer.
{ print nIndent spaces }
procedure XMLOutStreamObj.OutIndent;
  var i: integer;
  begin for i \leftarrow 1 to nIndent do OutChar(``\');
  end:
```

166 I/O WITH XML Mizar Parser $\S543$

543. When printing a start-tag to the file, we start by printing the indentation, then we increment the indentation, then we print the "<" followed by the name of the tag.

```
{ print '<' and the representation of fEl with indenting }
procedure XMLOutStreamObj.Out_XElStart(const fEl: string);
begin OutIndent; inc(nIndent); OutChar('<'); OutString(fEl);
end;
```

544. When we are done writing the attributes of a tag, we print the ">" to the file, and we also print a newline to the file.

```
{ close the attributes with '>' }
procedure XMLOutStreamObj.Out_XAttrEnd;
begin OutChar( '>'); OutNewLine;
end;
```

545. When we want to write the tag, but omit the attributes, we can do so.

```
{ no attributes expected }
procedure XMLOutStreamObj.Out_XElStart0(const fEl: string);
begin Out_XElStart(fEl); Out_XAttrEnd;
end;
```

546. For empty-element tags, we should close the tag with "/>", print a new line, then *decrement* the indentation since there are no children to the tag.

```
{ print '/>' with indenting }
procedure XMLOutStreamObj.Out_XElEnd0;
begin OutString('/>'); OutNewLine; dec(nIndent);
end:
```

547. When printing an empty-element tag without any attributes, we can combine the preceding functions together.

```
{ no attributes and elements expected }
procedure XMLOutStreamObj.Out_XEl1 (const fEl: string);
begin Out_XElStart(fEl); Out_XElEnd0;
end;
```

548. Printing end-tags should first decrement the indentation *before* printing the indentation to the file (so that the end-tag vertically aligns with the associated start-tag). Then we print "</" followed by the tag name and then ">". We should print a newline to the file, too.

```
{ close the fEl element using '</' }
procedure XMLOutStreamObj.Out_XElEnd(const fEl: string);
begin dec(nIndent); OutIndent; OutString('</'); OutString(fEl); OutChar('>'); OutNewLine;
end:
```

549. When printing one attribute to a tag, we need a whitespace printed (to separate the tag's name — or preceding attribute — from the current attribute being printed), followed by the attribute's name printed with an equality symbol, then enquoted the value of the attribute.

```
{ print one attribute key-value pair } procedure XMLOutStreamObj.Out_XAttr(const fAt, fVal: string); begin OutChar(´¬'); OutString(fAt); OutString(´=¬'); OutString(fVal); OutChar(¬¬'); end;
```

 $\S550$ Mizar Parser I/O WITH XML 167

550. When the value of an attribute is an integer, invoke IntToStr(fVal) to pretend it is a string value. Then printing out to a file an attribute with an integer value boils down to printing out the attribute with a string value.

```
{ print one attribute key-value pair, where value is integer }
procedure XMLOutStreamObj.Out_XIntAttr(const fAt: string; fVal: integer);
begin Out_XAttr(fAt, IntToStr(fVal));
end;
```

551. We can now just compose writing the start of a tag ($\S543$), followed by its attributes ($\S552$), and then close the empty-element tag ($\S546$).

```
procedure XMLOutStreamObj.Out_XElWithPos(const fEl: string; const fPos: Position);
begin Out_XElStart(fEl); Out_PosAsAttrs(fPos); Out_XElEnd0;
end;
```

552. We will want to treat a *position* (i.e., the line and column) as two attributes. We print this out using $Out_PosAsAttrs$. We rely on the XMLDict's XMLAttrName for standardizing the name for the line and column.

```
procedure XMLOutStreamObj.Out_PosAsAttrs(const fPos: Position);
begin Out_XIntAttr(XMLAttrName[atLine], fPos.Line);
Out_XIntAttr(XMLAttrName[atCol], fPos.Col);
end:
```

553. We print a quoted attribute, leveraging printing attributes out to the file (§549). We just need to escape the XML string (§516).

```
 \begin{array}{ll} \textbf{procedure} \ XMLOutStreamObj.Out\_XQuotedAttr(\textbf{const} \ fAt, fVal: \ string); \\ \textbf{begin} \ Out\_XAttr(fAt, QuoteStrForXML(fVal)); \\ \textbf{end}; \end{array}
```

168

Vocabulary file dictionaries

```
554. Mizar works with vocabulary files (suffixed with .voc) for introducing new identifiers.
\langle \text{ dicthan.pas } 554 \rangle \equiv
  ⟨GNU License 4⟩
unit dicthan;
  interface
  uses mobjects;
  (Public constants for dicthan.pas 555)
  type SymbolCounters = array [`A`..`Z'] of word;
    SymbolIntSeqArr = array ['A' .. 'Z'] of IntSequence;
     (Class declarations for dicthan.pas 556)
     (Public function declarations for dicthan.pas 557)
  implementation
  uses mizenv, xml_inout, xml_dict;
  ⟨Implementation for dicthan.pas 558⟩
  end.
       We recall from Adam Grabowski, Artur Kornilowicz, and Adam Naumowicz's "Mizar in a Nutshell"
(§4.3, doi:10.6092/issn.1972-5787/1980), the various prefixes for vocabulary file entries:
  - G for structures

    K for left-functor brackets

  - L for right-functor brackets
  - M for modes
  - O for functors
  - R for predicates
  - U for selectors

    V for attributes

\langle \text{Public constants for dicthan.pas } 555 \rangle \equiv
const
  StandardPriority = 64;
  AvailableSymbols = [ `G`, `K`, `L`, `M`, `O`, `R`, `U`, `V`];
This code is used in section 554.
       There are only three classes in the dictionary handling module. We have an abstraction for a symbol
appearing in a vocabulary file, a sort of "checksum" for the counts of symbols appearing in a vocabulary file.
and a dictionary associating to each article name (string) a collection of symbols.
\langle \text{Class declarations for dicthan.pas } 556 \rangle \equiv
  \langle \text{Symbol for vocabulary 562} \rangle;
  ⟨Abstract vocabulary object declaration 571⟩;
  ⟨Vocabulary object declaration 573⟩;
This code is used in section 554.
```

```
⟨Public function declarations for dicthan.pas 557⟩ ≡
function GetPrivateVoc(const fName: string): PVocabulary;
function GetPublicVoc (const fName: string; var fVocFile: text ): PVocabulary;
procedure LoadMmlVcb (const aFileName: string; var aMmlVcb: MStringList);
procedure StoreMmlVcb(const aFileName: string; const aMmlVcb: MStringList);
procedure StoreMmlVcbX (const aFileName: string; const aMmlVcb: MStringList);
This code is used in section 554.
       We can test if an entry in the dictionary is valid. Remember, only functor symbols can have a priority
associated with it (and a priority is a number between 0 and 255 = 2^8 - 1, inclusive).
  Also remember, that a symbol in a dictionary entry cannot have whitespaces in it.
  define delete\_prefix \equiv Delete(lLine, 1, 1)
\langle \text{Implementation for dicthan.pas } 558 \rangle \equiv
function Is ValidSymbol (const aLine: string): boolean;
  var lLine: string; lKind: char; lPriority, lPos, lCode: integer;
  begin IsValidSymbol \leftarrow false; lLine \leftarrow TrimString(aLine);
  (Initialize lKind, but exit if dictionary line contains invalid symbol 559);
  delete\_prefix;
  case lKind of
  '0': (Check if functor symbol is valid 560);
  R: (Check if predicate symbol is valid 561);
  othercases begin if Pos(` \_ `, lLine) > 0 then exit;
    IsValidSymbol \leftarrow true;
    end;
  endcases;
  end:
See also sections 563, 567, 572, 574, 578, 580, and 581.
This code is used in section 554.
      An "invalid" line in the dictionary file would be empty lines (whose length is less than one), and lines
which do not start with a valid prefix. At the end of this chunk, the lKind should be initialized to the prefix
of the line.
\langle Initialize lKind, but exit if dictionary line contains invalid symbol 559\rangle \equiv
  if length(lLine) \le 1 then exit;
  lKind \leftarrow lLine[1];
  if \neg(lKind \in AvailableSymbols) then exit
This code is used in section 558.
```

560. Recall the specification for *Val* sets *lCode* to zero for success, and the nonzero values store the index where the string is not a numeric value.

We copy the identifier (as determined from the start of the line until, but not including, the index of the first space in the line) and throw away everything after the first whitespace.

When the identifier for the functor symbol is not an empty string and the priority can be determined unambiguously, then the functor symbol entry is valid. Otherwise it is invalid.

```
⟨ Check if functor symbol is valid 560⟩ ≡ begin IsValidSymbol \leftarrow true; lPos \leftarrow Pos(`\_`, lLine); if lPos \neq 0 then begin { Parse priority for symbol } val(TrimString(Copy(lLine, lPos, length(lLine))), lPriority, lCode); lLine \leftarrow TrimString(Copy(lLine, 1, lPos - 1)); IsValidSymbol \leftarrow (lCode = 0) \land (lLine \neq ``); end; end
```

561. A predicate entry in the dictionary file should not include a priority, nor should it include any whitespaces. This is the criteria for a valid predicate symbol entry in the dictionary.

We enforce this by finding the first "" character in the line. If there is one, then we trim both sides of the line (removing leading and trailing whitespace). We should have no more spaces in the line. If there is a space, then it is an invalid predicate symbol.

```
⟨ Check if predicate symbol is valid 561⟩ ≡
begin lPos ← Pos(´¬, lLine);
if lPos ≠ 0 then { lLine contains a space }
begin lLine ← TrimString(Copy(lLine, lPos, length(lLine)));
if Pos(´¬, lLine) > 0 then exit;
end;
Is ValidSymbol ← true;
end
```

This code is used in section 558.

This code is used in section 556.

This code is used in section 558.

562. TSymbol. These are used in kernel/accdict.pas. The *Kind* is its one-letter kind (discussed in §555), and *Repr* is its lexeme. For functors, its priority is stored as its *Prior*.

The "infinitive" appears to be only used for predicates.

```
 \langle \operatorname{Symbol for vocabulary 562} \rangle \equiv \\ PSymbol = \uparrow TSymbol; \\ TSymbol = \operatorname{object }(MObject) \\ Kind: char; \\ Repr, Infinitive: string; \\ Prior: byte; \\ \operatorname{constructor }Init(fKind: char; fRepr, fInfinitive: string; fPriority: byte); \\ \operatorname{constructor }Extract(\operatorname{const }aLine: string); \\ \operatorname{function }SymbolStr: string; \\ \operatorname{constructor }Load(\operatorname{var }aText: text); \\ \operatorname{procedure }Store(\operatorname{var }aText: text); \\ \operatorname{destructor }Done; virtual; \\ \operatorname{end} \\
```

563. Constructor. Given the "kind", its "representation" and "infinitive", and its priority (as a number between 0 and 255), we can construct a symbol.

```
\langle Implementation for dicthan.pas 558\rangle += constructor TSymbol.Init(fKind:char;fRepr,fInfinitive:string;fPriority:byte); begin <math>Kind \leftarrow fKind;Repr \leftarrow fRepr;Prior \leftarrow fPriority;Infinitive \leftarrow ~~; end;
```

564. Constructor. When we want to extract a symbol from a line in the dictionary file, care must be taken for functors (since they may contain an explicit priority) and for predicates. Predicates have an undocumented feature to allow "infinitives", so an acceptable predicate line in a dictionary may look like

```
Rpredicate infinitive
```

Although what Mizar does with infinitives, I do not know...

```
constructor TSymbol.Extract(\mathbf{const}\ aLine:\ string);
var lPos, lCode:\ integer;\ lRepr:\ string;
begin Kind \leftarrow aLine[1];\ Repr \leftarrow TrimString(Copy(aLine,2,length(aLine)));\ Prior \leftarrow 0;
Infinitive \leftarrow ``;
case Kind of
`0`:\ \mathbf{begin}\ lPos \leftarrow Pos(`u`,Repr);\ Prior \leftarrow StandardPriority;
if lPos \neq 0 then \langle Initialize explicit priority for functor entry in dictionary 566\rangle;
end;
`R`:\ \mathbf{begin}\ lPos \leftarrow Pos(`u`,Repr);
if lPos \neq 0 then \langle Initilize explicit infinitive for a predicate entry in dictionary 565\rangle;
end;
endcases;
end;
```

565. Predicates can have an optional infinitive, separated from the lexeme by a single whitespace. It remains unclear what Mizar uses predicate infinitives for, but it is a feature. This is written out to the .vcx file, according to xml_dict.pas.

Note that there are 4 predicates with infinitives in Mizar:

- (1) jumps_in (infinitive: jump_in) occurs in the article AMISTD_1
- (2) halts_in (infinitive: halt_in) occurs in the article EXTPRO_1
- (3) refers (infinitive: refer) occurs in the article SCMFSA7B
- (4) destroys (infinitive: destroy) occurs in the article SCMFSA7B

 \langle Initilize explicit infinitive for a predicate entry in dictionary 565 \rangle \equiv **begin** $lRepr \leftarrow Repr$; $Repr \leftarrow TrimString(Copy(lRepr, 1, lPos - TrimString))$

```
 \begin{array}{l} \mathbf{begin} \ lRepr \leftarrow Repr; \ Repr \leftarrow ``; \ Repr \leftarrow TrimString(Copy(lRepr, 1, lPos-1)); \\ Infinitive \leftarrow TrimString(Copy(lRepr, lPos+1, length(lRepr))); \\ \mathbf{end} \end{array}
```

This code is used in section 564.

566. Functors with explicit priorities require parsing that priority. It is assumed that a single whitespace separates the lexeme from the priority.

```
\langle \text{Initialize explicit priority for functor entry in dictionary 566} \rangle \equiv 
\mathbf{begin} \ lRepr \leftarrow Repr; \ Repr \leftarrow ``; 
val(TrimString(Copy(lRepr, lPos + 1, length(lRepr))), Prior, lCode); 
Repr \leftarrow TrimString(Copy(lRepr, 1, lPos - 1)); 
\{ \text{Store the lexeme} \} 
\mathbf{end}
```

This code is used in section 564.

end;

567. Serialize symbols. We can serialize a *TSymbol* object, which produces the sort of entry we'd expect to find in a dictionary. So we would have the symbol kind, the lexeme, and optional data (non-default priorities for functors, infinitives for predicates).

```
\langle \text{Implementation for dicthan.pas } 558 \rangle + \equiv
function TSymbol.SymbolStr: string;
  var lStr, lIntStr: string;
  begin lStr \leftarrow Kind + Repr;
  case Kind of
   \texttt{`O'}: \mathbf{if} \ Prior \neq StandardPriority \ \mathbf{then}
       begin Str(Prior, lIntStr); lStr \leftarrow lStr + ' ' + lIntStr;
  'R': if Infinitive \neq ' then lStr \leftarrow lStr + ' + Infinitive;
  endcases;
  SymbolStr \leftarrow lStr;
  end;
568. Given a text (usually the contents of a vocabulary file), we read in a line. When the line is a
nonempty string, we initialize the lexeme representation, priority, and infinitives. Then, when the dictionary
entry describes a valid symbol (\S558), we populate the fields of the TSymbol.
constructor TSymbol.Load(var aText : text);
  var lDictLine: string;
  begin ReadLn(aText, lDictLine); lDictLine \leftarrow TrimString(lDictLine);
  if length(lDictLine) = 0 then exit;
  Repr \leftarrow ``; Prior \leftarrow 0; Infinitive \leftarrow ``;
  if Is ValidSymbol(lDictLine) then Extract(lDictLine);
```

569. Storing a TSymbol in a file amounts to writing its serialization ($\S 567$) to the file.

```
procedure TSymbol.Store(var aText : text);
  begin WriteLn(aText, SymbolStr);
  end;
```

570. Destructor. We just reset the lexeme and infinitive strings to be empty strings.

```
destructor TSymbol.Done;
begin Repr ← ´´; Infinitive ← ´´;
end;
```

571. Abstract vocabulary objects. This is used in kernel/impobjs.pas. We recall (§554) that the *Symbol Counters* are just an enumerated type consisting of a single uppercase Latin Letter.

```
⟨ Abstract vocabulary object declaration 571⟩ ≡
AbsVocabularyPtr = ↑AbsVocabularyObj;
AbsVocabularyObj = object (MObject)
fSymbolCnt: SymbolCounters;
constructor Init;
destructor Done; virtual;
end
```

This code is used in section 556.

```
We only have the constructor and destructor for abstract vocabulary objects.
\langle \text{Implementation for dicthan.pas } 558 \rangle + \equiv
constructor Abs Vocabulary Obj. Init;
  begin FillChar(fSymbolCnt, SizeOf(fSymbolCnt), 0);
  end:
destructor Abs Vocabulary Obj. Done;
  begin end;
      Vocabulary objects. A "vocabulary object" is just a collection of PSymbols read in from a
vocabulary file.
  These are also used in kernel/accdict.pas.
\langle Vocabulary object declaration 573 \rangle \equiv
  PVocabulary = \uparrow TVocabulary;
  TVocabulary = \mathbf{object} \ (AbsVocabularyObj)
    Reprs: MCollection;
    constructor Init;
    constructor ReadPrivateVoc(const aFileName: string);
    constructor Load Voc (var a Text : text);
    procedure Store Voc (const aFileName: string; var aText: text);
    destructor Done; virtual;
  end
This code is used in section 556.
574. Constructor (Empty vocabulary). We can construct the empty vocabulary by just initializing
the underlying collection.
\langle \text{Implementation for dicthan.pas } 558 \rangle + \equiv
constructor TVocabulary.Init;
  begin FillChar(fSymbolCnt, SizeOf(fSymbolCnt), 0); Reprs.Init(10, 10);
  end;
575. Destructor. We only need to free up the underlying collection.
destructor TVocabulary.Done;
  begin Reprs.Done;
  end:
      Constructor. We can read from a private vocabulary file.
constructor TVocabulary.ReadPrivateVoc(const aFileName: string);
  var lDict: text; lDictLine: string; lSymbol: PSymbol;
  begin Init; Assign(lDict, aFileName);
  without\_io\_checking(reset(lDict));
  if ioresult \neq 0 then exit; { file is not ready to be read, bail out! }
  while \neg seekEOF(lDict) do \langle Read line into vocabulary from dictionary file 577\rangle;
  Close(lDict);
  end;
```

Mizar Parser

When reading dictionary lines into a vocabulary file, we skip over blank lines. Further, we only read valid entries into the vocabulary.

```
\langle Read line into vocabulary from dictionary file 577\rangle \equiv
  begin readln(lDict, lDictLine); lDictLine \leftarrow TrimString(lDictLine);
  if length(lDictLine) > 1 then { if dictionary line is not blank }
    begin lSymbol \leftarrow new(PSymbol, Extract(lDictLine));
    if IsValidSymbol(lDictLine) then { add the symbol }
       begin inc(fSymbolCnt[lSymbol\uparrow.Kind]); Reprs.Insert(lSymbol); end;
    end;
  end
```

This code is used in section 576.

174

Constructor. We can read in the vocabulary from a file. If I am not mistaken, this is usually from mml.vct. We have the first line look like "G3 K0 L0 M1 O7 R2 U4 V6", which enumerates the number of different types of definitions appearing in an article.

```
\langle \text{Implementation for dicthan.pas } 558 \rangle + \equiv
constructor TVocabulary.LoadVoc(var aText: text);
  var i, lSymbNbr, lNbr: integer; lKind, lDummy, c: Char;
  begin lSymbNbr \leftarrow 0; (Count lNbr the number of dictionary entries for an article 579);
  ReadLn(aText); Reprs.Init(10, 10);
  for i \leftarrow 1 to lSymbNbr do
    begin Reprs.Insert(new(PSymbol, Load(aText)));
    end;
  end;
```

Since the first line counts the different sorts of definitions appearing in the article, we can parse the numbers, then add them up. This initializes the fSymbolcCnt entry for c.

```
\langle \text{ Count } lNbr \text{ the number of dictionary entries for an article 579} \rangle \equiv
  for c \leftarrow \text{`A' to 'Z' do}
     if c \in AvailableSymbols then
        begin Read(aText, lKind, lNbr, lDummy); fSymbolCnt[c] \leftarrow lNbr; Inc(lSymbNbr, fSymbolCnt[c]);
        end
```

This code is used in section 578.

Storing a dictionary entry. This appends to a .vct file the entries for an article. Specifically, this is just the "#ARTICLE" and then the counts of the different kinds of definitions.

```
\langle Implementation for dicthan.pas 558 \rangle + \equiv
procedure TVocabulary.StoreVoc (const aFileName: string; var aText: text);
  var i: Byte; c: Char;
     begin WriteLn(aText, `#`, aFileName);
     for c \leftarrow \text{`A' to 'Z' do}
       if c \in AvailableSymbols then Write(aText, c, fSymbolCnt[c], ``\');
     WriteLn(aText);
     for i \leftarrow 0 to Reprs.Count - 1 do PSymbol(Reprs.Items \uparrow [i]) \uparrow .Store(aText);
     end;
```

581. Miscellaneous public-facing functions.

```
⟨Implementation for dicthan.pas 558⟩ +≡ function GetPrivateVoc(\mathbf{const}\ fName:\ string): PVocabulary; var lName:\ string; begin lName \leftarrow fName; if ExtractFileExt(lName) = ^ then \ lName \leftarrow lName + ^ .voc^ ; if \neg MFileExists(lName) then begin GetPrivateVoc \leftarrow \mathbf{nil};\ exit; end; GetPrivateVoc \leftarrow new(PVocabulary, ReadPrivateVoc(lName)); end:
```

582. Reading mml.vct entries. The \$MIZFILES/mml.vct file contains all the vocabularies concatenated together into one giant vocabulary file. It uses lines prefixed with "#" followed by the article name to separate the vocabularies from different files. We search for the given article name (stored in the *fName* argument). When we find it, we construct the Vocabulary object (§578).

```
function GetPublicVoc (const fName: string; var fVocFile: text): PVocabulary; var lLine: string; begin GetPublicVoc \leftarrow nil; reset(fVocFile); while \neg eof(fVocFile) do
   begin readln(fVocFile, lLine);
   if (length(lLIne) > 0) \land (lLine[1] = `#`) \land (copy(lLine, 2, length(lLine)) = fName) then begin GetPublicVoc \leftarrow new(PVocabulary, LoadVoc(fVocFile)); exit; end; end; end;
```

583. Reading from mml.vct. This function is used by libtools/checkvoc.dpr and in a couple user tools. In those other functions, they pass \$MIZFILES/mml.vct as the value for *aFileName*. This procedure will then populate the *aMmlVcb* file associating to each article name its vocabulary.

```
procedure LoadMmlVcb (const aFileName: string; var aMmlVcb: MStringList ); var lFile: text; lDummy: char; lDictName: string; r: Integer; begin FileExam(aFileName); Assign(lFile, aFileName); Reset(lFile); { initialize file for reading } aMmlVcb.Init(1000); aMmlVcb.fSorted \leftarrow true; while \neg eof(lFile) do begin ReadLn(lFile, lDummy, lDictName); r \leftarrow aMmlVcb.AddObject(lDictName, new(PVocabulary, LoadVoc(lFile))); end; Close(lFile); end;
```

584. Storing a vocabulary delegates much work ($\S580$). However, since *fCount* is not initialized, I am uncertain how this works, exactly... Furthermore, this function is not used anywhere in Mizar.

```
procedure StoreMmlVcb (const aFileName: string; const aMmlVcb: MStringList); var lFile: text; i: Integer; begin Assign(lFile, aFileName); Rewrite(lFile); with aMmlVcb do for i \leftarrow 0 to fCount - 1 do PVocabulary(fList\uparrow[i].fObject)\uparrow.StoreVoc(fList\uparrow[i].fString\uparrow, lFile); Close(lFile); end;
```

Mizar Parser

 $VCXfile.Out_XElEnd(XMLElemName[elSymbols])$

This code is used in section 585.

176

equivalent to the previous function. **procedure** StoreMmlVcbX(**const** aFileName: string; **const** aMmlVcb: MStringList); var i, s: Integer; c: char; VCXfile: XMLOutStreamPtr; **begin** $VCXfile \leftarrow new(XMLOutStreamPtr, OpenFile(aFileName));$ $VCXfile.Out_XElStart0(XMLElemName[elVocabularies]);$ with aMmlVcb do for $i \leftarrow 0$ to fCount - 1 do with $PVocabulary(fList\uparrow[i].fObject)\uparrow$ do **begin** VCXfile.Out_XElStart(XMLElemName[elVocabulary]); $VCXfile.Out_XAttr(XMLAttrName[atName], fList\uparrow[i].fString\uparrow); VCXfile.Out_XAttrEnd;$ ⟨ Write vocabulary counts to XML file 586⟩; ⟨ Write symbols to vocabulary XML file 587⟩; $VCXfile.Out_XElEnd(XMLElemName[elVocabulary]);$ $VCXfile.Out_XElEnd(XMLElemName[elVocabularies]); dispose(VCXfile,Done);$ end; **586.** We write out the counts of each kind of definition appearing in the article. \langle Write vocabulary counts to XML file 586 $\rangle \equiv$ { Kinds } for $c \leftarrow \text{`A' to `Z' do}$ if $c \in AvailableSymbols$ then **begin** VCXfile.Out_XElStart(XMLElemName[elSymbolCount]); $VCXfile.Out_XAttr(XMLAttrName[atKind], c);$ $VCXfile.Out_XIntAttr(XMLAttrName[atNr], fSymbolCnt[c]); VCXfile.Out_XElEnd0;$ end This code is used in section 585. **587.** We write out each symbol appearing in the article's vocabulary. $\langle \text{Write symbols to vocabulary XML file 587} \rangle \equiv$ { Symbols } $VCXfile.Out_XElStart0(XMLElemName[elSymbols]);$ for $s \leftarrow 0$ to Reprs.Count - 1 do with $PSymbol(Reprs.Items[s]) \uparrow do$ **begin** VCXfile.Out_XElStart(XMLElemName[elSymbol]); VCXfile.Out_XAttr(XMLAttrName[atKind], Kind); $VCXfile.Out_XAttr(XMLAttrName[atName], QuoteStrForXML(Repr));$ case Kind of '0': VCXfile.Out_XIntAttr(XMLAttrName[atPriority], Prior); R: if $Infinitive \neq$ then $VCXfile.Out_XAttr(XMLAttrName[atInfinitive], Infinitive);$ **end**; VCXfile.Out_XElEnd0; end:

Like StoreMmlVcb, this function is not used anywhere in Mizar. This appears to produce the XML-

§588 Mizar Parser SCANNER 177

File 16

Scanner

588. The scanner.pas file contains the *MTokeniser* and the *MScanner*.

It is worth noting: if we want to extend Mizar to support Unicode, then we would want to hack this file accordingly. Or create a utf8scanner module, whichever. This scanner class is built specifically to work with ASCII characters, specifically accepting printable characters and the space ("") characters as valid input.

```
 \langle \text{scanner.pas 588} \rangle \equiv \\ \langle \text{GNU License 4} \rangle \\ \text{unit } \textit{scanner}; \\ \text{interface}; \\ \text{uses } \textit{errhan}, \textit{mobjects}; \\ \text{const } \textit{MaxLineLength} = 80; \\ \textit{MaxConstInt} = 2147483647; \quad \{=2^{31}-1, \text{ maximal signed 32-bit integer}\} \\ \langle \text{Type declarations for scanner 589} \rangle \\ \text{implementation} \\ \text{uses } \textit{mizenv}, \textit{librenv}, \textit{mconsole}, \textit{xml\_dict}, \textit{xml\_inout}; \\ \langle \text{Implementation for scanner.pas 590} \rangle \\ \text{end} \; . \\ \text{See also section 719}.
```

589. Note that a *LexemRec* is really a standardized token. I was always raised to believe that a "lexeme" refers to the literal text underlying a token.

```
⟨Type declarations for scanner 589⟩ ≡

type ASCIIArr = array [chr(0) .. chr(255)] of byte;

LexemRec = record Kind: char;

Nr: integer;
end;
⟨Token object class 591⟩;
⟨Tokens collection class 593⟩;
⟨MToken object class 601⟩;
⟨MTokeniser class 604⟩;
⟨MScanner object class 632⟩;

This code is used in section 588.
```

178 SCANNER Mizar Parser §590

590. The "default allowed" characters are the 10 decimal digits, the 26 uppercase Latin letters, the 26 lowercase Latin letters, and the underscore ("_") character.

```
\langle Implementation for scanner.pas 590\rangle \equiv
\mathbf{var}\ DefaultAllowed\colon AsciiArr =
 { '_' allowed in identifiers by default! }
 See also sections 592, 594, 595, 596, 599, 600, 602, 603, 605, 608, 609, 613, 628, 629, 631, 633, 642, 643, 644, 645, and 646.
This code is used in section 588.
    Tokens object. A token contains a lexeme, but it extends an MStr object.
\langle \text{ Token object class 591} \rangle \equiv
 TokenPtr = \uparrow TokenObj;
 TokenObj = \mathbf{object} (MStrObj)
   fLexem: LexemRec;
   constructor Init(aKind : char; aNr : integer ; const aSpelling: string);
 end
This code is used in section 589.
    The constructor for a token requires its kind (functor, mode, predicate, etc.), and its internal
"number", as well as its raw lexeme aSpelling.
\langle Implementation for scanner.pas 590\rangle + \equiv
constructor TokenObj.Init(aKind: char; aNr: integer; const aSpelling: string);
 begin fLexem.Kind \leftarrow aKind; fLexem.Nr \leftarrow aNr; fStr \leftarrow aSpelling;
```

end;

§593 Mizar Parser COLLECTIONS OF TOKENS 179

Section 16.1. COLLECTIONS OF TOKENS

 $\mathbf{end};$

593. We can populate a token collection from a dictionary file, or we can start with an empty collection. We can save our collection to a file. We can also insert (or "collect") a new token into the collection.

```
\langle Tokens collection class 593\rangle \equiv
  TokensCollection = object (MSortedStrList)
    fFirstChar: array [chr(30) ... chr(255)] of integer;
    constructor InitTokens:
    constructor LoadDct(const aDctFileName: string);
    procedure SaveDct(const aDctFileName: string);
    procedure SaveXDct(const aDctFileName: string);
    function CollectToken(const aLexem: LexemRec; const aSpelling: string): boolean;
  end
This code is used in section 589.
      Construct empty token collection.
\langle Implementation for scanner.pas 590\rangle + \equiv
constructor TokensCollection.InitTokens;
  begin Init(100);
  end;
595. Insert. If the collection already contains the token described by aLexem, then we just free up the
memory allocated for the token (avoid duplicates). Otherwise, we insert the token.
\langle Implementation for scanner.pas 590\rangle + \equiv
function TokensCollection.CollectToken(const aLexem: LexemRec; const aSpelling: string): boolean;
  var k: integer; lToken: TokenPtr;
  begin lToken \leftarrow new(TokenPtr, Init(aLexem.Kind, aLexem.Nr, aSpelling));
  if Search(lToken, k) then { already contains token? }
    begin CollectToken \leftarrow false; dispose(lToken, Done)
  else begin CollectToken \leftarrow true; Insert(lToken)
```

596. Load a dictionary. We open the dictionary ".dct" file (expects the file name to be lacking that extension), and construct an empty token collection. Then we iterate through the dictionary, reading each line, forming a new token, then inserting it into the collection.

The ".dct" file contains all the identifiers from articles referenced in the environ part of an article, and it will always have the first 148 lines be for reserved keywords. The format for a ".dct" file consists of lines of the form

```
\langle kind \rangle \langle number \rangle_{\sqcup} \langle name \rangle
```

The "kind" is a single byte, the *number* is an integer assigned for the identifier, and *name* is the lexeme (string literal) for the identifier. This also has an XML file for this same information, the ".dcx" file.

```
\langle Implementation for scanner.pas 590\rangle +\equiv constructor TokensCollection.LoadDct (const aDctFileName: string); var Dct: text; lKind, lDummy: AnsiChar; lNr: integer; lString: string; <math>i: integer; c: char; begin assign(Dct, aDctFileName + `.dct'); reset(Dct); InitTokens; \langle Load all tokens from the dictionary 597\rangle; close(Dct); \langle Index first character appearances among definitions 598\rangle; end:
```

Mizar Parser

We just iterate through the dictionary, constructing a new token for each line we read.

```
\langle \text{Load all tokens from the dictionary } 597 \rangle \equiv
  while \neg seekEof(Dct) do
     begin readln(Dct, lKind, lNr, lDummy, lString);
     Insert(new(TokenPtr, Init(char(lKind), lNr, lString)));
```

This code is used in section 596.

180

We index the first appearance of each leading character in a token.

```
\langle Index first character appearances among definitions 598\rangle \equiv
  for c \leftarrow chr(30) to chr(255) do fFirstChar[c] \leftarrow -1;
   for i \leftarrow 0 to Count - 1 do
      begin c \leftarrow TokenPtr(Items\uparrow[fIndex\uparrow[i]])\uparrow.fStr[1];
      if fFirstChar[c] = -1 then fFirstChar[c] \leftarrow i;
      end
```

This code is used in section 596.

599. We save a token collection to a ".dct" file. This appears to just produce the concatenation of the definition kind, the identifier number, then a whitespace separating it from the lexeme. Caution: this is not an XML format! For that, see SaveDctX.

```
\langle Implementation for scanner.pas 590\rangle + \equiv
procedure TokensCollection.SaveDct(const aDctFileName: string);
  var i: integer; DctFile: text;
  begin assign(DctFile, aDctFileName + `.dct`); rewrite(DctFile);
  for i \leftarrow 0 to Count - 1 do
     with TokenPtr(Items\uparrow[i])\uparrow, fLexem do writeln(DctFile, AnsiChar(Kind), Nr, \lnot_{\Box}, fStr);
  close(DctFile);
  end;
```

§600 Mizar Parser COLLECTIONS OF TOKENS 181

600. Save dictionary to XML file. The RNC (compact Relax NG Schema): Local dictionary for an article. The symbol kinds still use very internal notation.

```
elSymbols =
  attribute atAid {xsd:string}?,
  element elSymbols {
    element elSymbol {
      attribute atKind {xsd:string},
      attribute atNr {xsd:integer},
      attribute atName {xsd:integer}
  }
This creates the .dct file for an article.
\langle Implementation for scanner.pas 590\rangle + \equiv
procedure TokensCollection.SaveXDct(const aDctFileName: string);
  var lEnvFile: XMLOutStreamObj; i: integer;
  begin lEnvFile.OpenFile(aDctFileName);
  with lEnvFile do
    begin Out_XElStart(XMLElemName[elSymbols]); Out_XAttr(XMLAttrName[atAid], ArticleID);
    Out_XQuotedAttr(XMLAttrName[atMizfiles], MizFiles);
    Out_XAttrEnd; { print elSymbols start-tag }
    for i \leftarrow 0 to Count - 1 do { print children elSymbol elements }
      with TokenPtr(Items\uparrow[i])\uparrow, fLexem do
        begin Out_XElStart(XMLElemName[elSymbol]);
        Out\_XQuotedAttr(XMLAttrName[atKind], Kind); Out\_XIntAttr(XMLAttrName[atNr], Nr);
        Out\_XQuotedAttr(XMLAttrName[atName],fStr); Out\_XElEnd0;
    Out_XElEnd(XMLElemName[elSymbols]); { print elSymbols end-tag }
    end;
  lEnvFile.Done;
  end;
```

182 MIZAR TOKEN OBJECTS Mizar Parser $\S601$

Section 16.2. MIZAR TOKEN OBJECTS

601. This appears to be tokens for a specific file. An MToken extends a Token (§591).
⟨MToken object class 601⟩ ≡

MTokenPtr = ↑MTokenObj;

MTokenObj = object (TokenObj)

fPos: Position;

constructor Init(aKind : char; aNr : integer; const aSpelling: string; const aPos: Position);
end
This code is used in section 589.

602. Constructor. Construct a token. This might be a tad confusing, at least for me, because the lexeme is stored in the *fStr* field, whereas the standardized token is stored in the *fLexem* field.

We do not need to invoke the constructor for any ancestor class, because we just construct everything here. This seems like a bug waiting to happen...

```
\langle Implementation for scanner.pas 590\rangle +\equiv constructor MTokenObj.Init(aKind:char;aNr:integer; const aSpelling: string; const aPos: Position); begin fLexem.Kind \leftarrow aKind; fLexem.Nr \leftarrow aNr; fStr \leftarrow aSpelling; fPos \leftarrow aPos; end;
```

603. Token Kind constants. There are four kinds of tokens we want to distinguish: all valid tokens are either (1) numerals, or (2) identifiers. Then we also have (3) error tokens. But last, we have (4) end of text tokens

These are for identifying everything which is neither an identifier defined in the vocabulary files, nor a reserved keyword.

```
\langle \text{Implementation for scanner.pas } 590 \rangle + \equiv 
const Numeral = \text{`N'}; Identifier = \text{`I'}; ErrorSymbol = \text{`?'}; EOT = \text{`!'};
```

 $\S604$ Mizar Parser TOKENISER 183

Section 16.3. TOKENISER

604. The first step in lexical analysis is to transform a character stream into a token stream. The Tokeniser extends the MToken object ($\S601$), which in turn extends the Token object ($\S591$).

In particular, we should take a moment to observe the new fields. The *fPhrase* field is a segment of the input stream which is expected to start at a non-whitespace character.

The *SliceIt* function populates the *TokensBuf* and the *fIdents* fields from the *fPhrase* field. I cannot find where *fTokens* is populated.

Note that the MTokeniser is not, itself, used anywhere *directly*. It's extended in the *MScannObj* class, which is used in base/mscanner.pas (and in kernel/envhan.pas).

The contract for GetPhrase ensures the fPhrase will be populated with a string ending with a space (" \square ") character or it will be the empty string. Any class extending MTokeniser must respect this contract.

```
\langle MTokeniser class 604 \rangle \equiv
  MTokeniser = \mathbf{object} (MTokenObj)
    fPhrase: string;
    fPhrasePos: Position;
    fTokensBuf: MCollection;
    fTokens, fIdents: TokensCollection;
    constructor Init;
    destructor Done; virtual;
    procedure SliceIt; virtual;
    procedure GetToken; virtual;
    procedure GetPhrase; virtual;
    function EndOfText: boolean; virtual;
    function IsIdentifierLetter(ch : char): boolean; virtual;
    function IsIdentifierFirstLetter(ch : char): boolean; virtual;
    function Spelling(const a Token: LexemRec): string; virtual;
  end
```

This code is used in section 589.

605. Spelling boils down to three cases (c.f., types of tokens $\S603$): numerals, identifiers, and everything else. Numerals spell out the base-10 decimal expansion.

The other two cases boil down to finding the first matching token in the caller's collection of tokens with the same lexeme supplied as an argument, provided certain 'consistency' checks hold (the lexeme and token have the same Kind).

```
⟨Implementation for scanner.pas 590⟩ +≡
function MTokeniser.Spelling(const aToken: LexemRec): string;
var i: integer; s: string;
begin Spelling ← ´´;
if aToken.Kind = Numeral then
   begin Str(aToken.Nr, s); Spelling ← s; end
else if aToken.Kind = Identifier then ⟨Spell an identifier for the MTokeniser 606⟩
   else ⟨Spell an error or EOF for the MTokeniser 607⟩;
end;
```

184 TOKENISER Mizar Parser §606

606. Spelling an identifier just needs to match the lexeme's number with the token's number. This finds the first matching token in the underlying collection, then terminates the function.

```
\langle Spell an identifier for the MTokeniser 606 \rangle \equiv begin for i \leftarrow 0 to fIdents.Count-1 do with TokenPtr(fIdents.Items\uparrow[i])\uparrow do if fLexem.Nr=aToken.Nr then begin Spelling \leftarrow fStr;\ exit end;
```

This code is used in section 605.

607. Spelling anything else for the tokeniser needs the kind and number of the lexeme to match those of the token. Again, this finds the first matching token in the underlying collection, then terminates the function.

```
⟨ Spell an error or EOF for the MTokeniser 607⟩ ≡ begin for i \leftarrow 0 to fTokens.Count - 1 do with TokenPtr(fTokens.Items\uparrow[i])\uparrow do if (fLexem.Kind = aToken.Kind) \land (fLexem.Nr = aToken.Nr) then begin Spelling \leftarrow fStr; exit end; end
```

This code is used in section 605.

608. Constructor. Initialising a tokeniser starts with a blank phrase and kind, with most fields set to zero.

```
⟨ Implementation for scanner.pas 590⟩ +≡ constructor MTokeniser.Init; begin fPos.Line \leftarrow 0; fLexem.Kind \leftarrow `¬¬; fPhrase \leftarrow `¬¬¬; fPhrasePos.Line \leftarrow 0; fPhrasePos.Col \leftarrow 0; fTokensBuf.Init(80,8); fTokens.Init(0); fIdents.Init(100); end;
```

609. Destructor. This chains to free up several fields, just invoking their destructors.

```
⟨ Implementation for scanner.pas 590 ⟩ +≡
destructor MTokeniser.Done;
begin fPhrase ← ´´; fTokensBuf.Done; fTokens.Done; fIdents.Done;
end;
```

610. Aside on ASCII separators. Note: chr(30) is the record separator in ASCII, and chr(31) is the unit separator. Within a group (or table), the records are separated with the "RS" (chr(30)). As far as unit separators, Lammer Bies explains (lammertbies.nl/comm/info/ascii-characters):

The smallest data items to be stored in a database are called units in the ASCII definition. We would call them field now. The unit separator separates these fields in a serial data storage environment. Most current database implementations require that fields of most types have a fixed length. Enough space in the record is allocated to store the largest possible member of each field, even if this is not necessary in most cases. This costs a large amount of space in many situations. The US control code allows all fields to have a variable length. If data storage space is limited—as in the sixties—this is a good way to preserve valuable space. On the other hand is serial storage far less efficient than the table driven RAM and disk implementations of modern times. I can't imagine a situation where modern SQL databases are run with the data stored on paper tape or magnetic reels...

We will introduce macros for the record separator and the unit separator, because Mizar's front-end uses them specifically for the following purposes:

- (1) lines longer than 80 characters will contain a record_separator character (§637);
- (2) all other invalid characters are replaced with the unit_separator character (c.f., §638).

```
define record\_separator \equiv chr(30)
define unit\_separator \equiv chr(31)
```

611. Example of zeroeth step ("getting a phrase") in tokenising. The *GetPhrase* function is left as an abstract method of the tokeniser, so it is worth discussing "What it is supposed to do" before getting to the tokenisation of strings.

Suppose we have the following snippet of Mizar:

```
theorem
for x being object
holds x= x;
```

This is "sliced up" into the following "phrases" (drawn in boxes) which are clustered by lines:

```
begin_|
theorem_|
for_| x_| being_| object_|
holds_| x=| x;_|
```

Observe that the "phrases" are demarcated by whitespaces (" $_{\sqcup}$ ") or linebreaks. This is the coarse "first pass" before we carve a "phrase" up into a token. A phrase contains at least one token, possibly multiple tokens (e.g., the phrase " $x=_{\sqcup}$ " contains the two tokens "x" and "=").

What is the contract for a "phrase"? A phrase is *guaranteed* to either be equal to " $_{\square}$ ", or it contains at least one token and it is *guaranteed* to end with a space " $_{\square}$ " character (ASCII code #32). Further, there are no other possible " $_{\square}$ " characters in a phrase *except* at the very end. A phrase is never an empty string.

The task is then to *slice up* each phrase into tokens.

186 TOKENISER Mizar Parser $\S612$

612. Tokenise a phrase. When a "phrase" has been loaded into the tokeniser (which is an abstract method implemented by its descendent classes), we tokenise it — "slice it up" into tokens, thereby populating the *fTokensBuf* tokens buffer. This is invoked as needed by the *GetToken* method (§629).

This function is superficially complex, but upon closer scrutiny it is fairly straightforward.

Also note, despite being marked as "virtual", this is not overridden anywhere in the Mizar program.

The contract ensures, barring catastrophe, the fLexem, fStr, and fPos be populated. **Importantly:** The fLexem's token type is one of the four kinds given in the constant section ($\S603$): Numeral, Identifier, ErrorSymbol, or EOT. What about the "reserved keywords" of Mizar? They are already present in the ".dct" file, which is loaded into the fTokens dictionary. So they will be discovered in step ($\S619$) in this procedure.

```
\langle \text{ Variables for slicing a phrase } 612 \rangle \equiv
lCurrChar: integer; { index in fPhrase for current position }
EndOfSymbol: integer;
EndOfIdent: integer; { index in fPhrase for end of identifier }
FoundToken: TokenPtr; { most recently found token temporary variable }
lPos: Position; { position for debugging purposes }
See also sections 615, 618, 621, 623, and 625.
This code is used in section 613.
      \langle Implementation for scanner.pas 590\rangle + \equiv
procedure MTokeniser.SliceIt;
  var \( \text{Variables for slicing a phrase 612} \)
  begin MizAssert(2333, fTokensBuf.Count = 0); { Requires: token buffer is empty }
  lCurrChar \leftarrow 1; \ lPos \leftarrow fPhrasePos;
  \langle \text{Slice pragmas 614} \rangle;
  while fPhrase[lCurrChar] \neq 1 do
     begin (Determine the ID 616);
     ⟨Try to find a dictionary symbol 619⟩;
     if EndOfSymbol < EndOfIdent then \langle Check identifier is not a number 622\rangle;
     if FoundToken \neq nil then
       with FoundToken↑ do
          begin lPos.Col \leftarrow fPhrasePos.Col + EndOfSymbol - 1;
         fTokensBuf.Insert(new(MTokenPtr, Init(fLexem.Kind, fLexem.Nr, fStr, lPos)));
         lCurrChar \leftarrow EndOfSymbol + 1; continue;
          end;
     \{ else FoundToken = nil \}
     (Whoops! We found an unknown token, insert a 203 error token 627);
     end;
  end;
614. We begin by slicing pragmas. This will insert the pragma into the tokens buffer.
  Note that the "$EOF" pragma indicates that we should treat the file as ending here. So we comply with
the request, inserting the EOT (end of text) token as the next token to be offered to the user.
\langle \text{Slice pragmas 614} \rangle \equiv
  if (lPos.Col = 1) \land (Pos("::$", fPhrase") = 1) then
     \textbf{begin}\ \textit{fTokensBuf}. Insert(new(\textit{MTokenPtr}, Init(`\_`, 0, copy(\textit{fPhrase}, 3, length(\textit{fPhrase}) - 3), lPos)));\\
     if copy(fPhrase, 1, 6) = :: \$EOF then
       fTokensBuf.Insert(new(MTokenPtr,Init(EOT,0,fPhrase,lPos)));
     exit
     end
```

This code is used in section 613.

 $\S615$ Mizar Parser TOKENISER 187

615. We take the longest possible substring consisting of identifier characters as a possible identifier. The phrase is guaranteed to contain at least one token, maybe more, so we just keep going until we have exhausted the phrase or found a non-identifier character.

Note that all invalid characters are transformed into the "unit character" (c.f., §638). We should treat any occurrence of them as an error.

At the end of this stage of our tokenising journey, for valid tokens, we should have EndOfIdent and IdentLength both initialized here.

```
 \langle \text{Variables for slicing a phrase } 612 \rangle +\equiv \\ \textit{IdentLength: integer};   \textbf{616.} \quad \langle \text{Determine the ID } 616 \rangle \equiv \\  \{ \text{1. attempt to determine the ID } \} \\  EndOfIdent \leftarrow \textit{lCurrChar}; \\  \textbf{if } \textit{IsIdentifierFirstLetter}(\textit{fPhrase}[\textit{EndOfIdent}]) \textbf{ then} \\  \textbf{while } (\textit{EndOfIdent} < \textit{length}(\textit{fPhrase})) \land \textit{IsIdentifierLetter}(\textit{fPhrase}[\textit{EndOfIdent}]) \textbf{ do} \\  \textit{inc}(\textit{EndOfIdent}); \\  \textit{IdentLength} \leftarrow \textit{EndOfIdent} - \textit{lCurrChar}; \\  \textbf{if } \textit{fPhrase}[\textit{EndOfIdent}] \leq \textit{unit\_separator} \textbf{ then} \\  \langle \text{Whoops! ID turns out to be invalid, insert an error token, then continue } 617 \rangle; \\  \textit{dec}(\textit{EndOfIdent})
```

617. Recall (§637), we treat record separators as indicating the line is "too long" (i.e., more than 80 characters long). So we insert a 201 "Too long source line" error. But anything else is treated as an invalid identifier error.

```
\langle Whoops! ID turns out to be invalid, insert an error token, then continue 617 \rangle \equiv begin lPos.Col \leftarrow fPhrasePos.Col + EndOfIdent - 1; if fPhrase[EndOfIdent] = record\_separator then fTokensBuf.Insert(new(MTokenPtr,Init(ErrorSymbol,200, ``,lPos))) else fTokensBuf.Insert(new(MTokenPtr,Init(ErrorSymbol,201, ``,lPos))); lCurrChar \leftarrow EndOfIdent + 1; continue; end
```

This code is used in section 616.

This code is used in section 613.

188 TOKENISER Mizar Parser $\S618$

618. We look at the current phrase and try to match against tokens found in the underlying dictionary. When we find a match, we check if there are *multiple* matches and return the last one (this reflects Mizar's "the last version of the notation is preferred"). We implement this matching scheme using an infinite loop. Note that this uses a "**repeat...until** false" loop, which is identical to "**while** true **do begin ...end**" loop. (I am tempted to introduce a macro just to have this loop "**repeat...until** end_of_time"...)

Recall ($\S250$), sorted lists have a field fIndex which is an array of indices (which are sorted while leaving the underlying array Items of data untouched).

Also, lToken, lIndex are used only in this code chunk. Here lToken is translated to an index of the underlying dictionary, so for clarity we introduce a macro to refer to the token directly. And lIndex is used as "the current character" index to compare the phrase to the token (indexed by lToken) as a match or not.

At the end of this chunk, if successful, then FoundToken will be set to a valid token pointer. Further, EndOfPhrase will be initialized.

A possible bug: what happens if we look through the entire phrase? We can't "look any farther" down the phrase, so shouldn't we throw an error? Or lazily read more characters? Or...something?

Never fear: this will never happen with Mizar's grammar. The "reserved words" are *always* separated from the other stuff by at least one whitespace.

Also we note the list of symbols is sorted lexicographically.

This appears to match the phrase with the longest possible matching entry in the list of symbols (it is "maximal munch").

```
define the_item(#) ≡ Items↑[fIndex↑[#]]
define the_token(#) ≡ TokenPtr(the_item(#))↑

⟨Variables for slicing a phrase 612⟩ +≡

EndOfPhrase: integer; {index in fPhrase for candidate token}

lIndex: integer; {index for fIndex entry}

lToken: integer; {index for entries in dictionary starting with the first character of the current token}

619. Reserved keywords and defined terms are loaded into the fTokens dictionary.

⟨Try to find a dictionary symbol 619⟩ ≡

EndOfPhrase ← lCurrChar; FoundToken ← nil; EndOfSymbol ← EndOfPhrase − 1;

{initialized for comparison}

lToken ← fTokens.fFirstChar[fPhrase[EndOfPhrase]]; inc(EndOfPhrase);

if (lToken ≥ 0) then

with fTokens do

borin |Index ← 2;
```

This code is used in section 613.

until false;

end

 $\S620$ Mizar Parser TOKENISER 189

If we have *lIndex* (the index of the current phrase) be longer than the lexeme of the current dictionary entry's lexeme, then we should populate FoundItem. \langle If we matched a dictionary entry, then initialize FoundToken 620 $\rangle \equiv$ if $lIndex > length(the_token(lToken).fStr)$ then { we matched the token } **begin** $FoundToken \leftarrow the_item(lToken)$; $EndOfSymbol \leftarrow EndOfPhrase - 1$; end This code is used in section 619. When the identifier is not a number, we insert an "identifier" token into the tokens buffer. $\langle \text{ Variables for slicing a phrase } 612 \rangle + \equiv$ lFailed: integer; { index of first non-digit character } I: integer; { index ranging over the raw lexeme string } lSpelling: string; { raw lexeme as a string } **622.** \langle Check identifier is not a number $622 \rangle \equiv$ **begin** $lSpelling \leftarrow copy(fPhrase, lCurrChar, IdentLength);$ $lPos.Col \leftarrow fPhrasePos.Col + EndOfIdent - 1;$ if $(ord(fPhrase[lCurrChar]) > ord(`0`)) \wedge (ord(fPhrase[lCurrChar]) \leq ord(`9`))$ then **begin** $lFailed \leftarrow 0$; { location of non-digit character } for $I \leftarrow 1$ to IdentLength - 1 do $\mathbf{if} \ (\mathit{ord}(\mathit{fPhrase}[\mathit{lCurrChar} + I]) < \mathit{ord}(\texttt{`O'})) \lor (\mathit{ord}(\mathit{fPhrase}[\mathit{lCurrChar} + I]) > \mathit{ord}(\texttt{`9'})) \ \mathbf{then}$ **begin** $lFailed \leftarrow I + 1$; break; end: if lFailed = 0 then { if all characters are digits } ⟨ Whoops! Identifier turned out to be a number! 626⟩; \langle Add token to tokens buffer and iterate 624 \rangle ; This code is used in section 613. We add an *Identifier* token to the tokens buffer. $\langle \text{Variables for slicing a phrase } 612 \rangle + \equiv$ *lIdent*: TokenPtr; **624.** \langle Add token to tokens buffer and iterate 624 $\rangle \equiv$ $lIdent \leftarrow new(TokenPtr, Init(Identifier, fIdents.Count + 1, lSpelling));$ if fIdents.Search(lIdent, I) then dispose(lIdent, Done)**else** fIdents.Insert(lIdent); fTokensBuf. $Insert(new(MTokenPtr, Init(Identifier, TokenPtr(fIdents.Items \uparrow [I]) \uparrow .fLexem.Nr, lSpelling,$ $lCurrChar \leftarrow EndOfIdent + 1$; continue This code is used in section 622.

190 TOKENISER Mizar Parser $\S625$

625. If we goofed and all the characters turned out to be digits (i.e., the identifier was a numeral after all), we should clean things up here. Observe we will end up *continue*-ing along the loop.

When the numeral token is larger than $MaxConstInt = 2^{31} - 1$ (the largest 32-bit integer, §588), then we should raise a "Too large numeral" 202 error token. If we wanted to support "arbitrary precision" numbers, then this should be modified.

We can either insert into the tokens buffer an error token (in two possible outcomes) or a numeral token (in the third possible outcome).

```
\langle \text{ Variables for slicing a phrase } 612 \rangle + \equiv
lNumber: longint;
J: integer;
626. Whoops! Identifier turned out to be a number! 626 \geq
  begin if IdentLength > length(IntToStr(MaxConstInt)) then {insert error token}
    begin fTokensBuf.Insert(new(MTokenPtr, Init(ErrorSymbol, 202, lSpelling, lPos)));
    lCurrChar \leftarrow EndOfIdent + 1; continue;
    end:
  lNumber \leftarrow 0; J \leftarrow 1;
  for I \leftarrow IdentLength - 1 downto 0 do
    begin lNumber \leftarrow lNumber + (ord(fPhrase[lCurrChar + I]) - ord("0")) * J; J \leftarrow J * 10;
  if lNumber > MaxConstInt then { insert error token }
    begin fTokensBuf.Insert(new(MTokenPtr, Init(ErrorSymbol, 202, lSpelling, lPos)));
    lCurrChar \leftarrow EndOfIdent + 1; continue;
    end; { insert numeral token }
  fTokensBuf.Insert(new(MTokenPtr,Init(Numeral,lNumber,lSpelling,lPos)));
  lCurrChar \leftarrow EndOfIdent + 1; continue;
This code is used in section 622.
627. If we have tokenised the phrase, but the token is not contained in the dictionary, then we should raise
a 203 error.
\langle Whoops! We found an unknown token, insert a 203 error token 627 \rangle \equiv
  lPos.Col \leftarrow fPhrasePos.Col + lCurrChar - 1;
  fTokensBuf.Insert(new(MTokenPtr,Init(ErrorSymbol,203,fPhrase[lCurrChar],lPos)));\ inc(lCurrChar)
This code is used in section 613.
       We have purely abstract methods which will invoke Abstract1 (\S183), which raises a runtime error.
\langle Implementation for scanner.pas 590\rangle + \equiv
procedure MTokeniser.GetPhrase;
  begin Abstract1;
  end;
function MTokeniser.EndOfText: boolean;
  begin Abstract1; EndOfText \leftarrow false;
function MTokeniser.IsIdentifierLetter(ch:char): boolean;
  begin Abstract1; IsIdentifierLetter \leftarrow false;
  end;
```

§629 Mizar Parser TOKENISER 191

629. Get a token. Getting a token from the tokeniser will check if we've exhausted the input stream (which tests if the kind of fLexem is EOT), and exit if we have.

Otherwise, it looks to see if we've got tokens left in the buffer. If so, just pop one and exit.

But when the token buffer is empty, we invoke the abstract method *GetPhrase* to read some of the input stream. If it turns out there's nothing left to read, then update the tokeniser to be in the "end of text" state.

When we have some of the input stream read into the *fPhrase* field, we tokenise it using the *SliceIt* function. Then we pop a token from the buffer of tokens.

This will populate fLexem, fStr, and fPos with the new token, lexeme, and position... but that's only because GetPhrase (§633) and SliceIt (§612) do the actual work.

```
⟨ Implementation for scanner.pas 590⟩ +≡
procedure MTokeniser.GetToken;
begin if fLexem.Kind = EOT then exit;
if fTokensBuf.Count > 0 then
   begin ⟨ Pop a token from the underlying tokens stack 630⟩;
   exit;
   end;
GetPhrase;
if EndOfText then
   begin fLexem.Kind ← EOT; fStr ← ´´; fPos ← fPhrasePos; inc(fPos.Col);
   exit;
   end;
SliceIt; ⟨ Pop a token from the underlying tokens stack 630⟩;
end;
```

630. Popping a token will update the lexeme, str, and position fields to be populated from the first item in the tokens buffer. Then it will free that item from the tokens buffer, shifting everything down by one.

```
\langle \text{ Pop a token from the underlying tokens stack } 630 \rangle \equiv fLexem \leftarrow MTokenPtr(fTokensBuf.Items\uparrow[0])\uparrow.fLexem; fStr \leftarrow MTokenPtr(fTokensBuf.Items\uparrow[0])\uparrow.fStr; fPos \leftarrow MTokenPtr(fTokensBuf.Items\uparrow[0])\uparrow.fPos; fTokensBuf.AtFree(0)
```

This code is used in sections 629 and 629.

631. Testing if the given character is an identifier character or not requires invoking the abstract method IsIdentifierLetter (§628).

```
\langle Implementation for scanner.pas 590\rangle +\equiv function MTokeniser.IsIdentifierFirstLetter(ch: char): boolean; begin IsIdentifierFirstLetter \leftarrow IsIdentifierLetter(ch); end;
```

192 SCANNER OBJECT Mizar Parser §632

Section 16.4. SCANNER OBJECT

This extends the Tokeniser class (§604). It is the only class extending the Tokeniser class. $\langle MScanner object class 632 \rangle \equiv$ $MScannPtr = \uparrow MScannObj;$ $MScannObj = \mathbf{object} \ (MTokeniser)$ Allowed: ASCIIArr; fSourceBuff: pointer; fSourceBuffSize: word;fSourceFile: text; fCurrentLine: string; **constructor** *InitScanning* (**const** *aFileName*, *aDctFileName*: *string*); **destructor** Done; virtual; **procedure** GetPhrase; virtual; **procedure** ProcessComment(fLine, fStart: integer; cmt: string); virtual; **function** EndOfText: boolean; virtual; **function** *IsIdentifierLetter*(*ch* : *char*): *boolean*; *virtual*; end

This code is used in section 589.

633. Get a phrase. We search through the lines for the "first phrase" (i.e., first non-whitespace character, which indicates the start of something interesting). Comments are thrown away as are Mizar pragmas.

This will update fCurrentLine as needed, setting it to the next line in the input stream buffer. It will assign a copy of the phrase to the field fPhrase, as well as update the fPhrasePos.

There is a comment in Polish, "uzyskanie pierwszego znaczacego znaku", which Google translates as: "obtaining the first significant sign". This seemed like a natural "chunk" of code to study in isolation.

The contract for GetPhrase ensures the fPhrase will be populated with a string ending with a space (" $_{\sqcup}$ ") character, or it will be the empty string (when the end of text has been encountered).

```
⟨ Implementation for scanner.pas 590⟩ +≡
procedure MScannObj.GetPhrase;
const Prohibited: ASCIIArr = ⟨ Characters prohibited by MScanner 634⟩;
var i,k: integer;
begin fPhrasePos.Col ← fPhrasePos.Col + length(fPhrase) - 1;
⟨ Find the first significant 'sign' 635⟩;
for i ← fPhrasePos.Col to length(fCurrentLine) do
    if fCurrentLine[i] = ´ □ ´ then break;
fPhrase ← Copy(fCurrentLine, fPhrasePos.Col, i - fPhrasePos.Col + 1);
end;
```

 $\S634$ Mizar Parser SCANNER OBJECT 193

634. The prohibited ASCII characters are everything *NOT* among the follow characters:

```
...! " # $ % & ' ( ) * + , - . / : ; < = > ? @
[ \ ] ^ _ ' { | } ~ 0 1 2 3 4 5 6 7 8 9
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
```

The reader will observe these are all the "graphic" ASCII characters, plus the space ("□") character.

This code is used in section 633.

635. Note that the *fCurrentLine* will end with a whitespace, when we have not consumed the entire underlying input stream.

```
⟨ Find the first significant 'sign' 635⟩ ≡
while fCurrentLine[fPhrasePos.Col] = ´¬¬ do
begin if fPhrasePos.Col ≥ length(fCurrentLine) then ⟨ Populate the current line 636⟩;
inc(fPhrasePos.Col);
end
```

This code is used in section 633.

636. Now, populating the current line requires a bit of work. We ensure the end of the current line will end with a space character (""), which will guarantee the loop iteratively consumes all empty lines in the file.

Once we arrive at a non-space character, we will break the loop containing this chunk of code. If we have exhausted the underlying input stream, then we will have *EndOfText* be true. Should that occur, we exit the function.

```
⟨ Populate the current line 636⟩ ≡

begin if EndOfText then exit;

inc(fPos.Line); inc(fPhrasePos.Line); readln(fSourceFile, fCurrentLine);

⟨ Scan for pragmas, and exit if we found one 640⟩;

⟨ Skip comments 641⟩;

⟨ Trim whitespace from the right of the current line 639⟩;

⟨ Replace every invalid character in current line with the unit character 638⟩;

fCurrentLine \leftarrow fCurrentLine + `\_`;

if \neg LongLines then

if length(fCurrentLine) > MaxLineLength then ⟨ Replace end of long line with record separator 637⟩;

{ Assert: we have fCurrentLine end in "\_"⟩

fPhrasePos.Col \leftarrow 0; fPos.Col \leftarrow 0;
end
```

This code is used in section 635.

194 SCANNER OBJECT Mizar Parser §637

637. When we have excessively long lines, and we have not enabled "long line mode", then we just delete everything after MaxLineLength + 1 and set MaxLineLength - 1 to the record separator (which is rejected by the Mizar lexer) and the last character in the line to the space character.

```
 \langle \, \text{Replace end of long line with record separator } \, 637 \, \rangle \equiv \\ \mathbf{begin} \  \, delete(fCurrentLine, MaxLineLength + 1, length(fCurrentLine)); \\ fCurrentLine[MaxLineLength - 1] \leftarrow record\_separator; \\ fCurrentLine[MaxLineLength] \leftarrow `\ '\ '\ '; \\ \mathbf{end}
```

This code is used in section 636.

638. In particular, if we every encounter an "invalid" character, then we just replace it with the "unit separator" character.

```
\langle \text{Replace every invalid character in current line with the unit character } 638 \rangle \equiv  for k \leftarrow 1 to length(fCurrentLine) - 1 do if Prohibited[fCurrentLine[k]] > 0 then fCurrentLine[k] \leftarrow unit\_separator This code is used in section 636.
```

639. We will trim whitespace from the right of the current line at least twice.

```
 \langle \text{ Trim whitespace from the right of the current line } 639 \rangle \equiv k \leftarrow length(fCurrentLine); \\ \textbf{while } (k>0) \wedge (fCurrentLine[k] = `$\bot$') \textbf{ do } dec(k); \\ delete(fCurrentLine, k+1, length(fCurrentLine))
```

This code is used in sections 636 and 640.

640. Pragmas in Mizar are special comments which start a line with "::\$". They are useful for naming theorems ("::\$N $\langle name \rangle$ "), or toggling certain phases of the Mizar checker. This will process the comment (§642).

Since pragmas are important, we treat it as a token (and not a comment to be thrown away).

Note: if you try to invoke a pragma, but do not place it at the start of a line, then Mizar will treat it like a comment.

```
⟨ Scan for pragmas, and exit if we found one 640⟩ ≡ k \leftarrow Pos(`::$`,fCurrentLine); { Preprocessing directive } if (k = 1) then begin ProcessComment(fPhrasePos.Line,1,copy(fCurrentLine,1,length(fCurrentLine))); ⟨ Trim whitespace from the right of the current line 639⟩; fCurrentLine \leftarrow fCurrentLine + `\_\_`; fPhrase \leftarrow Copy(fCurrentLine,1,length(fCurrentLine)); <math>fPhrasePos.Col \leftarrow 1; fPos.Col \leftarrow 0; exit end
```

This code is used in section 636.

641. Scanning a comment will effectively replace the start of the comment ("::") up to and including the end of the line, with a single space. This will process the comment ($\S642$).

```
 \begin{split} &\langle \text{Skip comments } 641 \rangle \equiv \\ &k \leftarrow Pos(\texttt{`::`}, fCurrentLine); \quad \{ \text{Comment} \} \\ &\text{if } (k \neq 0) \text{ then} \\ &\text{ begin } ProcessComment(fPhrasePos.Line, k, copy(fCurrentLine, k, length(fCurrentLine))); \\ &delete(fCurrentLine, k+1, length(fCurrentLine)); \; fCurrentLine[k] \leftarrow \texttt{`u'}; \\ &\text{end} \end{split}
```

This code is used in section 636.

§642 Mizar Parser SCANNER OBJECT 195

```
642. "Processing a comment" really means skipping the comment.
```

```
⟨Implementation for scanner.pas 590⟩ +≡ procedure MScannObj.ProcessComment(fLine, fStart : integer; cmt : string); begin end;
```

643. Testing if the scanner has exhausted the input stream amounts to checking the current line has been completely read *and* the current source file has arrived at an *t*exttteof state.

```
\langle \text{Implementation for scanner.pas } 590 \rangle + \equiv function MScannObj.EndOfText: boolean; begin EndOfText \leftarrow (fPhrasePos.Col \geq length(fCurrentLine)) \wedge eof(fSourceFile); end;
```

644. Testing if a character is an identifier letter amounts to testing if it is allowed (i.e., not disallowed).

```
\langle Implementation for scanner.pas 590\rangle +\equiv function MScannObj.IsIdentifierLetter(ch:char): boolean; begin <math>IsIdentifierLetter \leftarrow Allowed[ch] \neq 0; end;
```

645. Constructor. The only way to construct a scanner. This expects an article to be read in aFileName and a dictionary to be loaded (aDctFileName, loaded with §596). The buffer size for reading aFileName is initially #4000.

```
⟨ Implementation for scanner.pas 590⟩ +≡
constructor MScannObj.InitScanning(const aFileName, aDctFileName: string);
begin inheritedInit; Allowed ← DefaultAllowed; fTokens.LoadDct(aDctFileName);
assign(fSourceFile, aFileName); fSourceBuffSize ← #4000; getmem(fSourceBuff, fSourceBuffSize);
settextbuf(fSourceFile, fSourceBuff↑, #4000); reset(fSourceFile); fCurrentLine ← ´¬¬; GetToken;
end;
```

646. Destructor. We must remember to close the source file, free the buffer, close the lights, and lock the doors.

```
⟨Implementation for scanner.pas 590⟩ +≡
destructor MScannObj.Done;
begin close(fSourceFile); FreeMem(fSourceBuff,fSourceBuffSize); fCurrentLine ← ´´; inheritedDone; end;
```

196 FORMAT Mizar Parser §647

File 17

Format

647. The first step towards disambiguating the meaning of identifiers is to use "formats". Recall from, e.g., Andrzej Trybulec's "Some Features of the Mizar Language" (ESPRIT Workshop, Torino, 1993; mizar.uwb.edu.pl/project/1933) that the "Format" describes with how many arguments a "Constructor Symbol" may be used. The basic formats:

- Predicates (lexeme, left arguments number, right arguments number)
- Modes (lexeme, arguments number) for "mode Foo of T_1, \ldots, T_n " where n is the arguments number
- Functors (lexeme, left arguments number, right arguments number)
- Bracket functors (left bracket lexeme, arguments number, right bracket lexeme)
- Selector (lexeme, 1)
- Structure (lexeme, arguments number) for generic structures over [arguments number] parameters
- Structure (lexeme, 1) for situations where we write "the [structure] of [term]"

We store these format information in XML files. See also Adam Grabowski, Artur Korniłowicz, and Adam Naumowicz's "Mizar in a Nutshell" (viz. §2.3, doi:10.6092/issn.1972-5787/1980) for a little more discussion about formats.

```
\langle format.pas 647 \rangle \equiv
  ⟨GNU License 4⟩
unit _formats;
  interface
  uses mobjects, scanner, dicthan, xml_inout;
    ⟨ Declare classes for _formats.pas 649⟩
  function CompareFormats(aItem1, aItem2: Pointer): Integer;
  function In_Format(fInFile: XMLInStreamPtr): MFormatPtr;
    ⟨Global variables (_formats.pas) 648⟩
  implementation
  uses errhan, xml_dict, xml_parser
      mdebug , info end_mdebug;
    ⟨Implementation for _formats.pas 650⟩
  end.
648.
\langle \text{Global variables (\_formats.pas) } 648 \rangle \equiv
var gFormatsColl: MFormatsList; gPriority: BinIntFunc; gFormatsBase: integer;
This code is used in section 647.
```

§649 Mizar Parser FORMAT 197

649. Broadly speaking, there are only 3 types of "formats": prefix formats, infix formats, bracket-like formats. These are viewed as "subclasses" of a base *MFormat* object.

We will want to collect the formats from articles referenced by the environment of an article being verified or parsed. This motivates the *MFormatList* object.

```
\langle \text{ Declare classes for \_formats.pas 649} \rangle \equiv
  \langle \text{ Declare } MFormat \text{ object } 651 \rangle;
     { TODO: add assertions that nr. of all format arguments is equal to the number of visible args
        (Visible) of a pattern }
  \langle \text{ Declare } MPrefixFormat \text{ object } 653 \rangle;
  \langle \text{ Declare } MInfixFormat \text{ object } 655 \rangle;
  \langle \text{ Declare } MBracketFormat \text{ object } 657 \rangle;
  \langle \text{ Declare } MFormatsList \text{ object } 665 \rangle;
This code is used in section 647.
        The presentation of the code is a bit disorganized from the perspective of pedagogy, so I am going to
re-organize for the sake of discussing it.
\langle \text{Implementation for \_formats.pas } 650 \rangle \equiv
   (Constructors for derived format classess 652)
   (Compare formats 659)
   (Implementation for MFormatsList 666)
   (Read formats from an XML input stream 677)
   ⟨Implement MFormatObj 678⟩
This code is used in section 647.
       Format base class. All format instances have a lexeme called its fSymbol. Recall that LexemeRec
(§589) is a normalized token using a single character to describe its kind, and an integer to keep track of it
(instead of relying on a raw string).
\langle \text{ Declare } MFormat \text{ object } 651 \rangle \equiv
  MFormatPtr = \uparrow MFormatObj;
  MFormatObj = \mathbf{object} \ (MObject)
     fSymbol: LexemRec;
     constructor Init(aKind : Char; aSymNr : integer);
     procedure Out_Format(\mathbf{var}\ fOutFile: XMLOutStreamObj;\ aFormNr: integer);
  end
This code is used in section 649.
        The constructor expects the "kind" of the object and its symbol number.
\langle Constructors for derived format classess 652 \rangle \equiv
constructor MFormatObj.Init(aKind : Char; aSymNr : integer);
  begin fSymbol.Kind \leftarrow aKind; fSymbol.Nr \leftarrow aSymNr;
  end:
See also sections 654, 656, and 658.
This code is used in section 650.
```

198 FORMAT Mizar Parser $\S653$

```
653. Prefix format object.
\langle \text{ Declare } MPrefixFormat \text{ object } 653 \rangle \equiv
  MPrefixFormatPtr = \uparrow MPrefixFormatObj;
  MPrefixFormatObj = \mathbf{object} (MFormatObj)
     fRightArqsNbr: byte;
     constructor Init(aKind : Char; aSymNr, aRArqsNbr : integer);
  end
This code is used in section 649.
654. Prefix formats track how many arguments are to the right of the prefix symbol.
\langle Constructors for derived format classess 652 \rangle + \equiv
constructor MPrefixFormatObj.Init(aKind: Char; aSymNr, aRArqsNbr: integer);
  begin fSymbol.Kind \leftarrow aKind; fSymbol.Nr \leftarrow aSymNr; fRightArgsNbr \leftarrow aRArgsNbr;
  end;
655. Infix format object.
\langle \text{ Declare } MInfixFormat \text{ object } 655 \rangle \equiv
  MInfixFormatPtr = \uparrow MInfixFormatObj;
  MInfixFormatObj = \mathbf{object} (MPrefixFormatObj)
     fLeftArgsNbr: byte;
     constructor Init(aKind : Char; aSymNr, aLArgsNbr, aRArgsNbr : integer);
  end
This code is used in section 649.
       And just as prefix symbols tracks the number of arguments to the right, infix symbols tracks the
number of arguments to both the left and right.
\langle Constructors for derived format classess 652 \rangle + \equiv
constructor MInfixFormatObj.Init(aKind: Char; aSymNr, aLArgsNbr, aRArgsNbr: integer);
  begin fSymbol.Kind \leftarrow aKind; fSymbol.Nr \leftarrow aSymNr; fLeftArgsNbr \leftarrow aLArgsNbr;
  fRightArgsNbr \leftarrow aRArgsNbr;
  end;
657. Bracket format object.
\langle \text{ Declare } MBracketFormat \text{ object } 657 \rangle \equiv
  MBracketFormatPtr = \uparrow MBracketFormatObj;
  MBracketFormatObj = \mathbf{object} (MInfixFormatObj)
     fRightSymbolNr: integer;
     fArgsNbr: byte;
     {f constructor}\ Init(aLSymNr,aRSymNr,aArgsNbr,aLArgsNbr,aRArgsNbr:integer);
  end
This code is used in section 649.
       \langle Constructors for derived format classess 652\rangle + \equiv
{f constructor}\ MBracketFormatObj.Init(aLSymNr, aRSymNr, aArgsNbr, aLArgsNbr, aRArgsNbr: integer);
  \textbf{begin} \ \textit{fSymbol.Kind} \leftarrow \texttt{`K'}; \ \textit{fSymbol.Nr} \leftarrow \textit{aLSymNr}; \ \textit{fRightSymbolNr} \leftarrow \textit{aRSymNr};
  fArgsNbr \leftarrow aArgsNbr; fLeftArgsNbr \leftarrow aLArgsNbr; fRightArgsNbr \leftarrow aRArgsNbr;
  end:
```

 $\S659$ Mizar Parser FORMAT 199

659. Ordering format objects. We need a *Compare* ordering function on formats. This is a lexicographic ordering on the (kind, number of right symbols, number of arguments, number of left symbols), more or less.

```
\langle \text{ Compare formats } 659 \rangle \equiv
function CompareFormats(aItem1, aItem2 : Pointer): Integer;
  begin CompareFormats \leftarrow 1;
  if MFormatPtr(aItem1)\uparrow.fSymbol.Kind < MFormatPtr(aItem2)\uparrow.fSymbol.Kind then
     CompareFormats \leftarrow -1
  else if MFormatPtr(aItem1)\uparrow.fSymbol.Kind = MFormatPtr(aItem2)\uparrow.fSymbol.Kind then
       \langle \text{ Compare symbols of the same kind } 660 \rangle;
  end:
This code is used in section 650.
660. We then check the indexing number of the symbol. When they are the same, we look at the next
"entry" in the tuple.
\langle Compare symbols of the same kind 660\rangle \equiv
  if MFormatPtr(aItem1)\uparrow.fSymbol.Nr < MFormatPtr(aItem2)\uparrow.fSymbol.Nr then
     CompareFormats \leftarrow -1
  else if MFormatPtr(aItem1)\uparrow.fSymbol.Nr = MFormatPtr(aItem2)\uparrow.fSymbol.Nr then
       (Compare same kinded symbols with the same number 661)
This code is used in section 659.
       The next "entry" in the tuple depends on the kind of symbols we are comparing. Selectors ('U')
are, at this point, identical (so we return zero). Note that 'J' is a historic artifact no longer used (in fact, I
cannot locate its meaning in the literature I possess).
  Structure ('G'), right functor brackets ('L'), modes ('M'), and attributes ('V') can be compared as prefix
  Functors ('0') and predicates ('R') can be compared as infix symbols.
  Left functor brackets ('K') can be compared first with bracket-specific characteristics, then as infix
\langle Compare same kinded symbols with the same number 661\rangle \equiv
  case MFormatPtr(aItem1)\uparrow.fSymbol.Kind of
  'J', 'U': CompareFormats \leftarrow 0;
  'G', 'L', 'M', 'V': (Compare prefix symbols 662);
  '0', 'R': (Compare infix symbols 664);
  'K': (Compare bracket symbols 663);
  endcases
This code is used in section 660.
       Comparing prefixing symbols, at this points, can only compare the number of arguments to the right.
\langle \text{ Compare prefix symbols 662} \rangle \equiv
  if MPrefixFormatPtr(aItem1)\uparrow.fRightArgsNbr < MPrefixFormatPtr(aItem2)\uparrow.fRightArgsNbr then
     CompareFormats \leftarrow -1
  else if MPrefixFormatPtr(aItem1)\uparrow.fRightArgsNbr = MPrefixFormatPtr(aItem2)\uparrow.fRightArgsNbr then
```

This code is used in section 661.

 $CompareFormats \leftarrow 0$

200 FORMAT Mizar Parser $\S 663$

663. Comparing bracket symbols first tries to compare the number of symbols to its right. If these are equal, then we try to compare the number of arguments. If these are equal, then we compare them "as if" they were infixing symbols.

 $\langle \, \text{Compare bracket symbols } \, 663 \, \rangle \equiv \\ \text{if } \, MBracketFormatPtr(aItem1) \uparrow .fRightSymbolNr < MBracketFormatPtr(aItem2) \uparrow .fRightSymbolNr \\ \text{then } \, CompareFormats \leftarrow -1 \\ \text{else if } \, MBracketFormatPtr(aItem1) \uparrow .fRightSymbolNr = MBracketFormatPtr(aItem2) \uparrow .fRightSymbolNr \\ \text{then} \\ \text{if } \, MBracketFormatPtr(aItem1) \uparrow .fArgsNbr < MBracketFormatPtr(aItem2) \uparrow .fArgsNbr \\ \text{then } \quad CompareFormats \leftarrow -1 \\ \text{else if } \, MBracketFormatPtr(aItem1) \uparrow .fArgsNbr = MBracketFormatPtr(aItem2) \uparrow .fArgsNbr \\ \text{then } \quad \langle \, \text{Compare infix symbols } \, 664 \, \rangle \\$

This code is used in section 661.

664. Comparing infixing symbols compares the number of arguments to the left. If these are equal, then we try to compare the number of arguments to the right. If these are equal, then we return 0.

```
\langle Compare infix symbols 664\rangle \equiv
```

- $\label{eq:linear_solution} \textbf{if} \ MInfixFormatPtr(aItem1) \uparrow .fLeftArgsNbr < MInfixFormatPtr(aItem2) \uparrow .fLeftArgsNbr \ \textbf{then} \\ CompareFormats \leftarrow -1$
- else if $MInfixFormatPtr(aItem1)\uparrow.fLeftArgsNbr = MInfixFormatPtr(aItem2)\uparrow.fLeftArgsNbr$ then if $MInfixFormatPtr(aItem1)\uparrow.fRightArgsNbr < MInfixFormatPtr(aItem2)\uparrow.fRightArgsNbr$ then $CompareFormats \leftarrow -1$
 - else if $MInfixFormatPtr(aItem1)\uparrow.fRightArgsNbr = MInfixFormatPtr(aItem2)\uparrow.fRightArgsNbr$ then $CompareFormats \leftarrow 0$

This code is used in sections 661 and 663.

 $\S665$ Mizar Parser LIST OF FORMATS 201

Section 17.1. LIST OF FORMATS

665. We have a collection of format objects managed by a *MFormatsList* object. There are two groups of public functions: "Lookup" functions (to find the format matching certain parameters), and "Collect" functions (to insert a new format).

```
\langle \text{ Declare } MFormatsList \text{ object } 665 \rangle \equiv
  MFormatsListPtr = \uparrow MFormatsList;
  MFormatsList = \mathbf{object} (MSortedList)
    constructor Init(ALimit : Integer);
    constructor LoadFormats(fName : string);
    procedure StoreFormats(fName : string);
    function LookUp_PrefixFormat(aKind : char; aSymNr, aArgsNbr : integer): integer;
    function LookUp\_FuncFormat(aSymNr, aLArgsNbr, aRArgsNbr : integer): integer;
    function LookUp\_BracketFormat(aLSymNr, aRSymNr, aArgsNbr, aLArgsNbr, aRArgsNbr: integer):
             integer;
    function LookUp_PredFormat(aSymNr, aLArgsNbr, aRArgsNbr: integer): integer;
    function CollectFormat(aFormat: MFormatPtr): integer;
    function CollectPrefixForm(aKind: char; aSymNr, aArgsNbr: integer): integer;
    function CollectFuncForm(aSymNr, aLArgsNbr, aRArgsNbr : integer): integer;
    function CollectBracketForm(aLSymNr, aRSymNr, aArgsNbr, aLArgsNbr, aRArgsNbr: integer):
    function CollectPredForm(aSymNr, aLArqsNbr, aRArqsNbr : integer): integer;
  end
```

This code is used in section 649.

666. We prefix format objects specified by its kind, its symbol number, and the number of arguments it expects.

When the format object is not found, then 0 will be returned. This is a standard convention in these functions to indicate the thing is missing.

```
⟨ Implementation for MFormatsList 666⟩ ≡ const PrefixFormatChars = [´M´, ´V´, ´U´, ´J´, ´L´, ´G´]; function MFormatsList.LookUp_PrefixFormat(aKind : char; aSymNr, aArgsNbr : integer): integer; var lFormat: MPrefixFormatObj; i: integer; begin MizAssert(3300, aKind ∈ PrefixFormatChars); lFormat.Init(aKind, aSymNr, aArgsNbr); if Find(@lFormat,i) then LookUp_PrefixFormat ← fIndex↑[i] + 1 else LookUp_PrefixFormat ← 0; end;
```

This code is used in section 650.

202 LIST OF FORMATS Mizar Parser $\S667$

667. Looking up an infix functor format (§655). This returns the *index* for the entry.

The contract here is rather confusing. What *should* occur is: if there is a functor symbol with the given left and right number of arguments, then return the index for the entry. Otherwise (when there is no functor symbol) return -1.

What happens instead is these values are incremented, so if the functor symbol with the given number of left and right arguments is contained in position k, then k+1 will be returned. If there is no such functor symbol, then 0 will be returned.

```
function MFormatsList.LookUp\_FuncFormat(aSymNr, aLArgsNbr, aRArgsNbr: integer): integer;
  var lFormat: MInfixFormatObj; i: integer;
  begin lFormat.Init(`O`, aSymNr, aLArgsNbr, aRArgsNbr);
  if Find(@lFormat, i) then LookUp\_FuncFormat \leftarrow fIndex \uparrow [i] + 1
  else LookUp\_FuncFormat \leftarrow 0;
  end;
668. Looking up a bracket.
function MFormatsList.LookUp_BracketFormat(aLSymNr, aRSymNr, aArgsNbr, aLArgsNbr,
         aRArgsNbr:integer):integer;
  var lFormat: MBracketFormatObj; i: integer;
  begin lFormat.Init(aLSymNr, aRSymNr, aArqsNbr, aLArqsNbr, aRArqsNbr);
  if Find(@lFormat, i) then LookUp\_BracketFormat \leftarrow fIndex \uparrow [i] + 1
  else LookUp\_BracketFormat \leftarrow 0;
  end;
669. Looking up a predicate.
function MFormatsList.LookUp\_PredFormat(aSymNr, aLArgsNbr, aRArgsNbr: integer): integer;
  var lFormat: MInfixFormatObj; i: integer;
  begin lFormat.Init('R', aSymNr, aLArgsNbr, aRArgsNbr);
  if Find(@lFormat, i) then LookUp\_PredFormat \leftarrow fIndex \uparrow [i] + 1
  else LookUp\_PredFormat \leftarrow 0;
  end;
670.
     Insert a format, if it's missing.
function MFormatsList.CollectFormat(aFormat: MFormatPtr): integer;
  var lFormatNr, i: integer;
  begin lFormatNr \leftarrow 0;
  if \neg Find(aFormat, i) then
    begin lFormatNr \leftarrow Count + 1; Insert(aFormat);
    end:
  CollectFormat \leftarrow lFormatNr;
  end;
```

 $\S671$ Mizar Parser LIST OF FORMATS 203

671.Inserting a bracket, if it is missing. Returns the format number for the format, whether it is missing or not. **function** MFormatsList.CollectBracketForm(aLSymNr, aRSymNr, aArgsNbr, aLArgsNbr, aRArqsNbr:integer):integer;**var** lFormatNr: integer; **begin** $lFormatNr \leftarrow LookUp_BracketFormat(aLSymNr, aRSymNr, aArqsNbr, aLArqsNbr, aRArqsNbr);$ if lFormatNr = 0 then **begin** $lFormatNr \leftarrow Count + 1$; Insert(new(MBracketFormatPtr, Init(aLSymNr, aRSymNr, aArgsNbr, aLArgsNbr, aRArgsNbr))); $CollectBracketForm \leftarrow lFormatNr;$ end: Inserting a functor format, if it is missing. This returns the format number for the functor (whether it is missing or not). **function** MFormatsList.CollectFuncForm(aSymNr, aLArgsNbr, aRArgsNbr: integer): integer; **var** lFormatNr: integer; **begin** $lFormatNr \leftarrow LookUp_FuncFormat(aSymNr, aLArgsNbr, aRArgsNbr);$ if lFormatNr = 0 then **begin** $lFormatNr \leftarrow Count + 1;$ Insert(new(MInfixFormatPtr, Init(`O`, aSymNr, aLArgsNbr, aRArgsNbr))); end; $CollectFuncForm \leftarrow lFormatNr;$ end: 673. Insert a prefix format if it is missing. Then return the format number for the prefix format, missing or not. **function** MFormatsList.CollectPrefixForm(aKind:char; aSymNr, aArgsNbr:integer): integer; var lFormatNr: integer; **begin** $lFormatNr \leftarrow LookUp_PrefixFormat(aKind, aSymNr, aArgsNbr);$ if lFormatNr = 0 then **begin** $lFormatNr \leftarrow Count + 1$; Insert(new(MPrefixFormatPtr, Init(aKind, aSymNr, aArqsNbr))); end: $CollectPrefixForm \leftarrow lFormatNr;$ end; 674. Insert a predicate format, if it is missing. Then return the format number, whether the predicate format is missing or not. **function** MFormatsList.CollectPredForm(aSymNr, aLArgsNbr, aRArgsNbr: integer): integer; **var** lFormatNr: integer; **begin** $lFormatNr \leftarrow LookUp_PredFormat(aSymNr, aLArgsNbr, aRArgsNbr);$ if lFormatNr = 0 then **begin** $lFormatNr \leftarrow Count + 1$; Insert(new(MInfixFormatPtr, Init(`R', aSymNr, aLArgsNbr, aRArgsNbr)));

 $CollectPredForm \leftarrow lFormatNr;$

end:

204 LIST OF FORMATS Mizar Parser $\S675$

```
Constructor. Construct the empty list of formats.
constructor MFormatsList.Init(ALimit: Integer);
  begin InitSorted(ALimit, CompareFormats);
  end:
676.
       Constructor. Parse an XML file for formats, and populate a format list object with the file's
contents.
constructor MFormatsList.LoadFormats(fName: string);
  var lEnvFile: XMLInStreamPtr; lValue: integer; lLex: LexemRec;
  begin InitSorted(100, CompareFormats); lEnvFile \leftarrow new(XMLInStreamPtr, OpenFile(fName));
  with lEnvFile↑ do
    begin NextElementState: XMLASSERT(nElName = XMLElemName[elFormats]):
    NextElementState;
    while \neg (nState = eEnd) \land (nElName = XMLElemName [elFormat]) do Insert(In\_Format(lEnvFile));
    gPriority.Init(10);
    while \neg (nState = eEnd) do
      begin XMLASSERT(nElName = XMLElemName[elPriority]);
       lLex.Kind \leftarrow GetAttr(XMLAttrName[atKind])[1];
       lLex.Nr \leftarrow GetIntAttr(XMLAttrName[atSymbolNr]); MizAssert(3300, lLex.Kind \in [`O', `L', `K']);
       lValue \leftarrow GetIntAttr(XMLAttrName[atValue]); gPriority.Assign(ord(lLex.Kind), lLex.Nr, lValue);
       AcceptEndState; NextElementState;
      end;
    end;
  dispose(lEnvFile, Done);
  end:
677. We can read exactly one format from an XML input stream.
\langle \text{Read formats from an XML input stream } 677 \rangle \equiv
function In_Format(fInFile : XMLInStreamPtr): MFormatPtr;
  var lLex: LexemRec; lArgsNbr, lLeftArgsNbr, lRightSymNr: integer;
  begin with fInFile↑ do
    begin lLex.Kind \leftarrow GetAttr(XMLAttrName[atKind])[1];
    lLex.Nr \leftarrow GetIntAttr(XMLAttrName[atSymbolNr]);
    lArgsNbr \leftarrow GetIntAttr(XMLAttrName[atArgNr]);
    case lLex.Kind of
     0', R': begin lLeftArqsNbr \leftarrow GetIntAttr(XMLAttrName[atLeftArqNr]);
       In\_Format \leftarrow new(MInfixFormatPtr, Init(lLex.Kind, lLex.Nr, lLeftArgsNbr,
           lArqsNbr - lLeftArqsNbr);
     \texttt{J'}, \texttt{U'}, \texttt{V'}, \texttt{G'}, \texttt{L'}, \texttt{M'}: In\_Format \leftarrow new(MPrefixFormatPtr, Init(lLex.Kind, lLex.Nr, lArgsNbr));
     \  \  \text{`K': begin } lRightSymNr \leftarrow GetIntAttr(XMLAttrName[atRightSymbolNr]);
       In\_Format \leftarrow new(MBracketFormatPtr, Init(lLex.Nr, lRightSymNr, lArgsNbr, 0, 0));
      end;
    othercases RunTimeError(2019);
    endcases:
    AcceptEndState; NextElementState;
    end:
  end;
```

This code is used in section 650.

 $\S678$ Mizar Parser LIST OF FORMATS 205

```
Conversely, we can print to an output stream an XML representation for a format object.
\langle \text{Implement } MFormatObj | 678 \rangle \equiv
procedure MFormatObj.Out_Format(var fOutFile: XMLOutStreamObj; aFormNr: integer);
  begin with fOutFile do
    begin Out_XElStart(XMLElemName[elFormat]); Out_XAttr(XMLAttrName[atKind], fSymbol.Kind);
    if aFormNr > 0 then Out\_XIntAttr(XMLAttrName[atNr], aFormNr);
    Out\_XIntAttr(XMLAttrName[atSymbolNr], fSymbol.Nr);
    case fSymbol.Kind of
    'J', 'U', 'V', 'G', 'L', 'M':
           Out\_XIntAttr(XMLAttrName[atArqNr], MPrefixFormatPtr(@Self) \uparrow fRightArqsNbr);
    `O`, `R`: with MInfixFormatPtr(@Self)↑ do
        begin Out\_XIntAttr(XMLAttrName[atArgNr], fLeftArgsNbr + fRightArgsNbr);
        Out\_XIntAttr(XMLAttrName[atLeftArgNr], fLeftArgsNbr);
        end:
    `K`: with MBracketFormatPtr(@Self)↑ do
        begin Out_XIntAttr(XMLAttrName[atArqNr], fArqsNbr);
        Out\_XIntAttr(XMLAttrName[atRightSymbolNr], fRightSymbolNr);
    othercases RuntimeError(3300);
    endcases;
    Out\_XElEnd\theta;
    end:
  end:
This code is used in section 650.
679. Given a list of formats, we can store them to an XML file using the previous function.
procedure MFormatsList.StoreFormats(fName : string);
  var lEnvFile: XMLOutStreamObj; z: integer;
  begin lEnvFile. OpenFile(fName);
  with lEnvFile do
    begin Out_XElStart0 (XMLElemName [elFormats]);
    for z \leftarrow 0 to Count - 1 do MFormatPtr(Items \uparrow [z]) \uparrow. Out\_Format(lEnvFile, z + 1);
    with qPriority do
      for z \leftarrow 0 to fCount - 1 do
        begin Out_XElStart(XMLElemName[elPriority]);
        Out\_XAttr(XMLAttrName[atKind], chr(fList\uparrow[z].X1));
        Out\_XIntAttr(XMLAttrName[atSymbolNr], fList\uparrow[z].X2);
        Out\_XIntAttr(XMLAttrName[atValue], fList\uparrow[z].Y); Out\_XElEnd0;
    Out_XElEnd(XMLElemName[elFormats]);
    end:
  lEnvFile.Done;
  end;
680. We clean up the formats collection and the priority. The gPriority is initialized and populated in
other functions. The qFormatsColl is used heavily in parseraddition.pas and a few other places.
procedure DisposeFormats;
  begin gFormatsColl.Done; gPriority.Done;
  end:
```

206 SYNTAX Mizar Parser §681

File 18

Syntax

```
681. This describes the syntax for the Mizar language, using expressions, subexpressions, blocks, and
"items" (statements).
  We will need to recall StackedObj from mobjects.pas (§307).
\langle \text{syntax.pas } 681 \rangle \equiv
  (GNU License 4)
unit syntax;
  interface
  uses mobjects, errhan; (Interface for syntax.pas 688)
  implementation
  uses mconsole
    mdebug, info end_mdebug;
    (Implementation for syntax.pas 683)
  end.
       The maximum number of "visible" arguments to an expression is set here, at 10.
682.
\langle \text{Public constants for syntax.pas } 682 \rangle \equiv
const MaxVisArqNbr = 10;
This code is used in section 688.
       The implementation for the abstract syntax of Mizar is rather uninteresting, since most of the methods
are abstract.
\langle \text{Implementation for syntax.pas } 683 \rangle \equiv
  (Subexpression constructor 714)
   (Subexpression destructor 715)
   Expression constructor 711
  (Subexpression procedures 718)
   Create a subexpression for an expression 712
   (Item object implementation 704)
  (Block object implementation 694)
  (Public procedures implementation for syntax.pas 684)
This code is used in section 681.
684. Destructor wrappers. We have a few public-facing procedures to free the global subexpression,
expression, etc., variables describing the state of the parser.
⟨Public procedures implementation for syntax.pas 684⟩ ≡
procedure KillSubexpression;
  begin if qSubexpPtr = nil then RunTimeError(2144)
  else dispose(gSubexpPtr, Done);
  end;
See also sections 685, 686, and 687.
This code is used in section 683.
```

§685 Mizar Parser SYNTAX 207

```
685.
⟨ Public procedures implementation for syntax.pas 684⟩ +≡
procedure KillExpression;
  begin if qExpPtr = nil then RunTimeError(2143)
  else dispose(gExpPtr, Done);
  end;
686.
      This method will not be used until we get to the parser, sadly. I am not sure why there are calls to
DisplayLine in KillItem and KillBlock, though.
  The KillItem is called in exactly two places: (1) Semicolon in parser.pas, (2) SchemeBlock, also in the
parser. (And KillBlock is called only in the parser, as well.)
⟨Public procedures implementation for syntax.pas 684⟩ +≡
procedure KillItem;
  begin if gItemPtr = nil then RunTimeError(2142)
  else begin gItemPtr \uparrow .Pop; dispose(gItemPtr, Done); end;
  DisplayLine(CurPos.Line, ErrorNbr);
  end;
687.
⟨ Public procedures implementation for syntax.pas 684⟩ +≡
procedure KillBlock;
  begin if gBlockPtr = nil then RunTimeError(2141)
  else begin gBlockPtr\uparrow.Pop; dispose(gBlockPtr,Done);
    end;
  DisplayLine(CurPos.Line, ErrorNbr);
  end;
688.
\langle \text{Interface for syntax.pas } 688 \rangle \equiv
  (Public constants for syntax.pas 682)
type (BlockKinds (syntax.pas) 692)
  ⟨ItemKinds (syntax.pas) 702⟩
  ⟨ExpKinds (syntax.pas) 709⟩
  ⟨Block object interface 693⟩;
  (Class declaration for Item object 703);
  (Subexpression object class 713);
  \langle \text{Expression class declaration } 710 \rangle;
  (Public procedures for syntax.pas 689)
  (Public variables for syntax.pas 690)
This code is used in section 681.
689. \langle \text{Public procedures for syntax.pas 689} \rangle \equiv
procedure KillBlock;
procedure KillItem;
procedure KillExpression;
procedure KillSubexpression;
This code is used in section 688.
```

208 SYNTAX Mizar Parser $\S 690$

690. These global public variables for syntax will be manipulated by the parser.

```
\langle \text{Public variables for syntax.pas } 690 \rangle \equiv  var gBlockPtr: BlockPtr = nil; gItemPtr: ItemPtr = nil; gExpPtr: ExpressionPtr = nil; gSubexpPtr: SubexpPtr = nil; This code is used in section 688.
```

 $\S691$ Mizar Parser BLOCK OBJECT 209

Section 18.1. BLOCK OBJECT

691. The Mizar language is block-structured, so we have a Block represent a sequence of statements contained within a block.

This is extended in parseraddition.pas.



Fig. 1. UML class diagram for Block object class.

692. There are about a dozen different kinds of blocks.

```
 \langle \ BlockKinds \ (\textbf{syntax.pas}) \ 692 \ \rangle \equiv \\ BlockKind = (blMain, blDiffuse, blHereby, blProof, blDefinition, blNotation, blRegistration, blCase, \\ blSuppose, blPublicScheme);
```

This code is used in section 688.

```
693. \langle Block object interface 693\rangle \equiv
  BlockPtr = \uparrow BlockObj;
  ItemPtr = \uparrow ItemObj;
  BlockObj = \mathbf{object} (StackedObj)
    nBlockKind: BlockKind;
    constructor Init(fBlockKind : BlockKind);
    procedure Pop; virtual;
                                 { inheritance }
    destructor Done; virtual;
    procedure StartProperText; virtual;
    procedure ProcessLink; virtual;
    procedure ProcessRedefine; virtual;
    procedure ProcessBegin: virtual:
    procedure ProcessPragma; virtual;
    procedure StartAtSignProof; virtual;
    procedure FinishAtSignProof; virtual;
    procedure FinishDefinition; virtual;
    procedure CreateItem(fItemKind : ItemKind); virtual;
    procedure CreateBlock(fBlockKind : BlockKind); virtual;
    procedure StartSchemeDemonstration; virtual;
    procedure FinishSchemeDemonstration; virtual;
  end
```

This code is used in section 688.

694. The constructor for a Block will initialize its Previous pointer to point at the global gBlockPtr instance.

```
⟨ Block object implementation 694⟩ ≡ constructor BlockObj.Init(fBlockKind : BlockKind); begin nBlockKind ← fBlockKind; Previous ← gBlockPtr; end;
See also sections 695, 696, 697, 698, 699, and 700.
This code is used in section 683.
```

210 BLOCK OBJECT Mizar Parser §695

```
Note that popping a block object is left for subclasses to handle.
\langle Block object implementation 694\rangle + \equiv
procedure BlockObj.Pop;
  begin end;
696. \langle Block object implementation 694\rangle + \equiv
destructor BlockObj.Done;
  begin gBlockPtr \leftarrow BlockPtr(Previous);
  end;
697. Abstract methods.
\langle Block object implementation 694\rangle + \equiv
procedure BlockObj.StartProperText;
  begin end;
procedure BlockObj.ProcessRedefine;
  begin end;
procedure BlockObj.ProcessLink;
  begin end;
procedure BlockObj.ProcessBegin;
  begin end;
procedure BlockObj.ProcessPragma;
  begin end;
procedure BlockObj.StartAtSignProof;
  begin end;
procedure BlockObj.FinishAtSignProof;
  begin end;
procedure BlockObj.FinishDefinition;
  begin end;
698. \langle Block object implementation 694\rangle + \equiv
procedure BlockObj.CreateItem(fItemKind: ItemKind);
  begin gItemPtr \leftarrow new(ItemPtr, Init(fItemKind));
  end;
      \langle Block object implementation 694\rangle + \equiv
procedure BlockObj.CreateBlock(fBlockKind: BlockKind);
  begin gBlockPtr \leftarrow new(BlockPtr, Init(fBlockKind));
  end;
700. More abstract methods.
\langle Block object implementation 694\rangle + \equiv
procedure BlockObj.StartSchemeDemonstration;
  begin end;
procedure BlockObj.FinishSchemeDemonstration;
  begin end;
```

 $\S701$ Mizar Parser ITEM OBJECTS 211

Section 18.2. ITEM OBJECTS

701. The class declaration for an *Item* object is depressingly long, with most of its virtual methods not used. The class diagram is worth drawing out.



Fig. 2. UML class diagram for Item object class.

702. Items are a tagged union, tagged by the "kind" of item.

```
 \begin{tabular}{l} $\langle ItemKinds\ (syntax.pas)\ 702 \rangle \equiv ItemKinds\ (syntax.pas)\ 702 \rangle \equiv ItemKinds\ (itIncorrItem,itDefinition,itSchemeBlock,itSchemeHead,itTheorem,itAxiom,itReservation,itCanceled,itSection,itRegularStatement,itChoice,itReconsider,itPrivFuncDefinition,itPrivPredDefinition,itConstantDefinition,itGeneralization,itLociDeclaration,itExistentialAssumption,itExemplification,itPerCases,itConclusion,itCaseBlock,itCaseHead,itSupposeHead,itAssumption,itCorrCond,itCorrectness,itProperty,itDefPred,itDefFunc,itDefMode,itDefAttr,itDefStruct,itPredSynonym,itPredAntonym,itFuncNotation,itModeNotation,itAttrSynonym,itAttrAntonym,itCluster,itIdentify,itReduction,itPropertyRegistration,itPragma); \end{tabular}
```

This code is used in section 688.

```
703. ⟨Class declaration for Item object 703⟩ ≡
ItemObj = object (StackedObj)
nItemKind: ItemKind;
constructor Init(fItemKind : ItemKind);
procedure Pop; virtual;
destructor Done; virtual;
⟨Method declarations for Item object 707⟩
end
```

This code is used in section 688.

704. It is particularly important to note, when constructing an *Item* object, the previous item will automatically be set to point to the global *qItem* variable.

```
⟨ Item object implementation 704⟩ ≡
constructor ItemObj.Init(fItemKind : ItemKind);
  begin nItemKind ← fItemKind; Previous ← gItemPtr;
  end;
procedure ItemObj.Pop;
  begin DisplayLine(CurPos.Line, ErrorNbr);
  end;
destructor ItemObj.Done;
  begin DisplayLine(CurPos.Line, ErrorNbr); gItemPtr ← ItemPtr(Previous);
  end;
See also sections 705 and 708.
This code is used in section 683.
```

212 ITEM OBJECTS Mizar Parser $\S705$

```
705. Creating an expression in an item is handled with this method. \langle Item object implementation 704\rangle +\equiv procedure ItemObj. CreateExpression(fExpKind: ExpKind); begin gExpPtr \leftarrow new(ExpressionPtr, Init(fExpKind)); end;
```

8706 Mizar Parser ITEM OBJECTS 213

706. Abstract methods. The methods of the *Item* class can be partitioned into two groups: those which will be implemented by a subclass, and those which will remain "empty" (i.e., whose body is just **begin end**).

```
\langle Methods overriden by extended Item class 706\rangle \equiv
procedure StartSentence: virtual:
procedure StartAttributes; virtual;
procedure FinishAntecedent; virtual;
procedure FinishConsequent; virtual;
procedure FinishClusterTerm; virtual;
procedure StartFuncIdentify; virtual;
procedure ProcessFuncIdentify; virtual;
procedure CompleteFuncIdentify; virtual;
procedure ProcessLeftLocus; virtual;
procedure ProcessRightLocus; virtual;
procedure StartFuncReduction; virtual;
procedure ProcessFuncReduction; virtual;
procedure FinishPrivateConstant; virtual;
procedure StartFixedVariables: virtual:
procedure ProcessFixedVariable; virtual;
procedure ProcessBeing: virtual:
procedure StartFixedSegment; virtual;
procedure FinishFixedSeament; virtual;
procedure FinishFixedVariables; virtual;
procedure StartAssumption; virtual;
procedure StartCollectiveAssumption; virtual;
procedure ProcessMeans; virtual;
procedure FinishOtherwise; virtual;
procedure StartDefiniens: virtual:
procedure FinishDefiniens; virtual;
procedure StartGuard; virtual;
procedure FinishGuard; virtual;
procedure ProcessEquals; virtual;
procedure StartExpansion; virtual;
procedure FinishSpecification; virtual;
procedure StartConstructionType; virtual;
procedure FinishConstructionType: virtual:
procedure StartAttributePattern; virtual;
procedure FinishAttributePattern; virtual;
procedure FinishSethoodProperties; virtual;
procedure StartModePattern; virtual;
procedure FinishModePattern; virtual;
procedure StartPredicatePattern; virtual;
procedure ProcessPredicateSymbol; virtual;
procedure FinishPredicatePattern; virtual;
procedure StartFunctorPattern; virtual;
procedure ProcessFunctorSymbol; virtual;
procedure FinishFunctorPattern; virtual;
procedure ProcessAttrAntonym; virtual;
procedure ProcessAttrSynonym; virtual;
procedure ProcessPredAntonym; virtual;
procedure ProcessPredSynonym; virtual;
```

214 ITEM OBJECTS Mizar Parser §706

```
procedure ProcessFuncSynonym; virtual;
procedure ProcessModeSynonym; virtual;
procedure StartVisible; virtual;
procedure Process Visible: virtual:
procedure FinishPrefix; virtual;
procedure ProcessStructureSymbol; virtual;
procedure StartFields; virtual;
procedure FinishFields; virtual;
procedure StartAggrPattSegment; virtual;
procedure ProcessField; virtual;
procedure FinishAggrPattSegment; virtual;
procedure ProcessSchemeName; virtual;
procedure StartSchemeSegment; virtual;
procedure StartSchemeQualification; virtual;
procedure FinishSchemeQualification; virtual;
procedure ProcessScheme Variable; virtual;
procedure FinishSchemeSegment; virtual;
procedure FinishSchemeThesis; virtual;
procedure FinishSchemePremise; virtual;
procedure StartReservationSegment; virtual;
procedure ProcessReservedIdentifier; virtual;
procedure FinishReservationSegment; virtual;
procedure StartPrivateDefiniendum; virtual;
procedure FinishLocusType: virtual:
procedure CreateExpression(fExpKind : ExpKind); virtual;
procedure StartPrivateConstant; virtual;
procedure StartPrivateDefiniens; virtual;
procedure FinishPrivateFuncDefinienition; virtual;
procedure FinishPrivatePredDefinienition; virtual;
procedure ProcessReconsideredVariable; virtual;
procedure FinishReconsideredTerm; virtual;
procedure FinishDefaultTerm; virtual;
procedure FinishCondition; virtual;
procedure FinishHypothesis; virtual;
procedure ProcessExemplifyingVariable; virtual;
procedure FinishExemplifyingVariable; virtual;
procedure StartExemplifyingTerm; virtual;
procedure FinishExemplifyingTerm; virtual;
procedure ProcessCorrectness; virtual;
procedure ProcessLabel; virtual;
procedure StartRegularStatement; virtual;
procedure ProcessDefiniensLabel; virtual;
procedure FinishCompactStatement; virtual;
procedure StartIterativeStep; virtual;
procedure FinishIterativeStep; virtual;
    { Justification }
procedure ProcessSchemeReference; virtual;
procedure ProcessPrivateReference; virtual;
procedure StartLibraryReferences; virtual;
procedure StartSchemeLibraryReference; virtual;
procedure ProcessDef; virtual;
```

 $\S706$ Mizar Parser ITEM OBJECTS 215

procedure ProcessTheoremNumber; virtual; procedure ProcessSchemeNumber; virtual; procedure StartJustification; virtual; procedure StartSimpleJustification; virtual; procedure FinishSimpleJustification; virtual;

See also section 1224.

This code is used in sections 707 and 1225.

216 ITEM OBJECTS Mizar Parser §707

```
707.
      \langle Method declarations for Item object 707 \rangle \equiv
  (Methods overriden by extended Item class 706)
procedure FinishClusterType; virtual;
procedure FinishSentence; virtual;
procedure FinishReconsidering; virtual;
procedure StartNewType; virtual;
procedure StartCondition; virtual;
procedure FinishChoice; virtual;
procedure FinishAssumption; virtual;
procedure StartEquals; virtual;
procedure StartOtherwise; virtual;
procedure StartSpecification; virtual;
procedure ProcessAttributePattern; virtual;
procedure StartDefPredicate; virtual;
procedure CompletePredAntonymByAttr; virtual;
procedure CompletePredSynonymByAttr; virtual;
procedure StartPredIdentify; virtual;
procedure ProcessPredIdentify; virtual;
procedure CompleteAttrIdentify; virtual;
procedure StartAttrIdentify; virtual;
procedure ProcessAttrIdentify: virtual:
procedure CompletePredIdentify; virtual;
procedure FinishFuncReduction; virtual;
procedure StartSethoodProperties; virtual;
procedure ProcessModePattern; virtual;
procedure StartPrefix; virtual;
procedure FinishVisible; virtual;
procedure FinishSchemeHeading; virtual;
procedure FinishSchemeDeclaration; virtual;
procedure StartSchemePremise: virtual:
procedure StartTheoremBody; virtual;
procedure FinishTheoremBody; virtual;
procedure FinishTheorem; virtual;
procedure FinishReservation; virtual;
procedure ProcessIterativeStep; virtual;
    \{ Justification \}
procedure StartSchemeReference; virtual;
procedure StartReferences; virtual;
procedure ProcessSch; virtual;
procedure FinishTheLibraryReferences; virtual;
procedure FinishSchLibraryReferences; virtual;
procedure FinishReferences; virtual;
procedure FinishSchemeReference; virtual;
procedure FinishJustification; virtual;
This code is used in section 703.
```

 $\langle \text{ Item object implementation } 704 \rangle + \equiv$ procedure ItemObj.StartAttributes; begin end; **procedure** *ItemObj.FinishAntecedent*; begin end; procedure ItemObj.FinishConsequent; begin end; **procedure** *ItemObj.FinishClusterTerm*; begin end; procedure ItemObj.FinishClusterType; begin end; **procedure** *ItemObj.StartSentence*; begin end; **procedure** *ItemObj.FinishSentence*; begin end; **procedure** *ItemObj* . *FinishPrivateConstant*; begin end; **procedure** *ItemObj.StartPrivateConstant*; begin end; **procedure** *ItemObj.ProcessReconsideredVariable*; begin end; ${\bf procedure}\ {\it ItemObj.FinishReconsidering};$ begin end; **procedure** *ItemObj.FinishReconsideredTerm*; begin end; procedure ItemObj.FinishDefaultTerm; begin end; procedure ItemObj.StartNewType; begin end; **procedure** *ItemObj.StartCondition*; begin end: **procedure** *ItemObj.FinishCondition*; begin end; ${\bf procedure}\ {\it ItemObj.FinishChoice};$ begin end; **procedure** *ItemObj.StartFixedVariables*; begin end; procedure ItemObj.StartFixedSegment; begin end; procedure ItemObj.ProcessFixedVariable; begin end; **procedure** *ItemObj.ProcessBeing*; begin end; **procedure** *ItemObj.FinishFixedSegment*; begin end; procedure ItemObj.FinishFixedVariables; begin end; **procedure** *ItemObj.StartAssumption*; begin end; **procedure** *ItemObj.StartCollectiveAssumption*; begin end; procedure ItemObj.FinishHypothesis;

218 ITEM OBJECTS Mizar Parser $\S708$

```
begin end;
procedure ItemObj.FinishAssumption;
  begin end;
procedure ItemObj.ProcessExemplifyingVariable;
  begin end;
procedure ItemObj.FinishExemplifyingVariable;
  begin end;
procedure ItemObj .StartExemplifyingTerm;
  begin end;
procedure ItemObj.FinishExemplifyingTerm;
  begin end;
procedure ItemObj.ProcessMeans;
  begin end;
procedure ItemObj.FinishOtherwise;
  begin end;
procedure ItemObj.StartDefiniens;
  begin end;
procedure ItemObj.FinishDefiniens;
  begin end;
procedure ItemObj.StartGuard;
  begin end;
\mathbf{procedure}\ \mathit{ItemObj.FinishGuard};
  begin end;
procedure ItemObj.StartOtherwise;
  begin end;
procedure ItemObj.ProcessEquals;
  begin end;
procedure ItemObj.StartEquals;
  begin end;
procedure ItemObj.ProcessCorrectness;
  begin end:
procedure ItemObj.FinishSpecification;
  begin end;
{\bf procedure}\ {\it ItemObj.FinishConstructionType};
  begin end;
procedure ItemObj.StartSpecification;
  begin end;
procedure ItemObj.StartExpansion;
  begin end;
procedure ItemObj . StartConstructionType;
  begin end;
procedure ItemObj.StartPredicatePattern;
  begin end;
procedure ItemObj.ProcessPredicateSymbol;
  begin end;
procedure ItemObj.FinishPredicatePattern;
  begin end;
procedure ItemObj.StartFunctorPattern;
  begin end;
procedure ItemObj.ProcessFunctorSymbol;
  begin end;
procedure ItemObj.FinishFunctorPattern;
```

```
begin end;
procedure ItemObj.ProcessAttrAntonym;
  begin end;
procedure ItemObj.ProcessAttrSynonym;
  begin end;
procedure ItemObj.ProcessPredAntonym;
  begin end;
procedure ItemObj.ProcessPredSynonym;
  begin end;
procedure ItemObj.ProcessFuncSynonym;
  begin end;
procedure ItemObj.CompletePredSynonymByAttr;
  begin end;
procedure ItemObj.CompletePredAntonymByAttr;
  begin end;
procedure ItemObj.ProcessModeSynonym;
  begin end;
procedure ItemObj.StartFuncIdentify;
  begin end;
procedure ItemObj.ProcessFuncIdentify;
  begin end;
{\bf procedure}\ {\it ItemObj.CompleteFuncIdentify};
  begin end;
procedure ItemObj.StartPredIdentify;
  begin end;
procedure ItemObj.ProcessPredIdentify;
  begin end;
procedure ItemObj.CompletePredIdentify;
  begin end;
procedure ItemObj.StartAttrIdentify;
  begin end:
procedure ItemObj.ProcessAttrIdentify;
  begin end;
procedure ItemObj.CompleteAttrIdentify;
  begin end;
procedure ItemObj.ProcessLeftLocus;
  begin end;
procedure ItemObj.ProcessRightLocus;
  begin end;
procedure ItemObj.StartFuncReduction;
  begin end;
procedure ItemObj.ProcessFuncReduction;
  begin end;
procedure ItemObj.FinishFuncReduction;
  begin end;
{\bf procedure}\ {\it ItemObj.StartSethoodProperties};
  begin end;
procedure ItemObj . FinishSethoodProperties;
  begin end;
procedure ItemObj.StartModePattern;
  begin end;
procedure ItemObj.ProcessModePattern;
```

220 ITEM OBJECTS Mizar Parser $\S708$

begin end; procedure ItemObj.FinishModePattern; begin end; procedure ItemObj.StartAttributePattern; begin end; **procedure** *ItemObj.ProcessAttributePattern*; begin end; **procedure** *ItemObj.FinishAttributePattern*; begin end; procedure ItemObj.StartDefPredicate; begin end; procedure ItemObj.StartVisible; begin end; **procedure** *ItemObj.ProcessVisible*; begin end; procedure ItemObj.FinishVisible; begin end; **procedure** *ItemObj.StartPrefix*; begin end; **procedure** *ItemObj.FinishPrefix*; begin end; ${\bf procedure}\ {\it ItemObj.ProcessStructureSymbol};$ begin end; procedure ItemObj.StartFields; begin end; **procedure** *ItemObj.FinishFields*; begin end; procedure ItemObj.StartAggrPattSegment; begin end; procedure ItemObj.ProcessField; begin end: **procedure** *ItemObj.FinishAggrPattSegment*; begin end; ${\bf procedure}\ {\it ItemObj.ProcessSchemeName};$ begin end; **procedure** *ItemObj.StartSchemeSegment*; begin end; **procedure** *ItemObj.ProcessSchemeVariable*; begin end; **procedure** *ItemObj.StartSchemeQualification*; begin end; **procedure** *ItemObj.FinishSchemeQualification*; begin end; **procedure** *ItemObj.FinishSchemeSegment*; begin end; procedure ItemObj.FinishSchemeHeading; begin end; **procedure** *ItemObj.FinishSchemeDeclaration*; begin end; **procedure** *ItemObj.FinishSchemeThesis*; begin end; **procedure** *ItemObj.StartSchemePremise*;

```
begin end;
procedure ItemObj.FinishSchemePremise;
  begin end;
procedure ItemObj.StartTheoremBody;
  begin end;
procedure ItemObj.FinishTheoremBody;
  begin end;
procedure ItemObj.FinishTheorem;
  begin end;
procedure ItemObj.StartReservationSegment;
  begin end;
procedure ItemObj.ProcessReservedIdentifier;
  begin end;
procedure ItemObj.FinishReservationSegment;
  begin end;
procedure ItemObj.FinishReservation;
  begin end;
procedure ItemObj.StartPrivateDefiniendum;
  begin end;
procedure ItemObj.FinishLocusType;
  begin end;
{\bf procedure}\ {\it ItemObj.StartPrivateDefiniens};
  begin end;
procedure ItemObj.FinishPrivateFuncDefinienition;
  begin end;
procedure ItemObj.FinishPrivatePredDefinienition;
  begin end;
procedure ItemObj.ProcessLabel;
  begin end;
procedure ItemObj.StartRegularStatement;
  begin end:
procedure ItemObj.ProcessDefiniensLabel;
  begin end;
procedure ItemObj.ProcessSchemeReference;
  begin end;
procedure ItemObj.StartSchemeReference;
  begin end;
procedure ItemObj.StartReferences;
  begin end;
procedure ItemObj.ProcessPrivateReference;
  begin end;
procedure ItemObj.StartLibraryReferences;
  begin end;
procedure ItemObj.StartSchemeLibraryReference;
  begin end;
procedure ItemObj.ProcessDef;
  begin end;
procedure ItemObj.ProcessSch;
  begin end;
procedure ItemObj.ProcessTheoremNumber;
  begin end;
procedure ItemObj.ProcessSchemeNumber;
```

222 ITEM OBJECTS Mizar Parser §708

begin end; procedure ItemObj.FinishTheLibraryReferences; begin end; **procedure** *ItemObj.FinishSchLibraryReferences*; begin end; procedure ItemObj.FinishReferences; begin end; procedure ItemObj.FinishSchemeReference; begin end; procedure ItemObj.StartJustification; begin end; **procedure** *ItemObj.FinishJustification*; begin end; procedure ItemObj.StartSimpleJustification; begin end; ${\bf procedure}\ {\it ItemObj.FinishSimpleJustification};$ begin end; procedure ItemObj.FinishCompactStatement; begin end; procedure ItemObj.StartIterativeStep; begin end; ${\bf procedure}\ {\it ItemObj.ProcessIterativeStep};$ begin end; procedure ItemObj.FinishIterativeStep; begin end;

 $\S709$ Mizar Parser EXPRESSIONS 223

Section 18.3. EXPRESSIONS

```
709.
\langle \text{ExpKinds (syntax.pas) } 709 \rangle \equiv
  ExpKind = (exNull, exType, exTerm, exFormula, exResType, exAdjectiveCluster);
This code is used in section 688.
710. \langle Expression class declaration 710 \rangle \equiv
  ExpressionPtr = \uparrow ExpressionObj;
  ExpressionObj = \mathbf{object} \ (MObject)
     nExpKind: ExpKind;
     constructor Init(fExpKind : ExpKind);
     procedure CreateSubexpression; virtual;
  end
This code is used in section 688.
711. Constructor.
\langle \text{Expression constructor } 711 \rangle \equiv
constructor ExpressionObj.Init(fExpKind : ExpKind);
  begin nExpKind \leftarrow fExpKind;
  end;
This code is used in section 683.
712. Observe that creating a subexpression (1) allocates a new SubexpPtr on the heap, and (2) mutates
the gSubexpPtr global variable.
\langle Create a subexpression for an expression 712 \rangle \equiv
procedure ExpressionObj.CreateSubexpression;
  begin gSubexpPtr \leftarrow new(SubexpPtr, Init);
  end;
This code is used in section 683.
```

224 SUBEXPRESSIONS Mizar Parser §713

Section 18.4. SUBEXPRESSIONS

```
713.
\langle Subexpression object class 713\rangle \equiv
  SubexpPtr = \uparrow SubexpObj;
  SubexpObj = \mathbf{object} (StackedObj)
    constructor Init;
    destructor Done; virtual;
     (Empty method declarations for SubexpObj 717)
  end
This code is used in section 688.
point to the global gSubexpPtr object.
```

714. Constructor. Importantly, constructing a new Subexp object will initialize its Previous field to

```
\langle Subexpression constructor 714\rangle \equiv
constructor SubexpObj.Init;
   \mathbf{begin}\ \mathit{Previous} \leftarrow \mathit{gSubexpPtr};
   end;
```

This code is used in section 683.

715. Destructor.

```
\langle Subexpression destructor 715\rangle \equiv
destructor SubexpObj.Done;
  begin gSubexpPtr \leftarrow SubexpPtr(Previous);
  end;
```

This code is used in section 683.

```
716.
      The remaining methods for subexpression objects are empty.
\langle Methods implemented by subclasses of SubexpObj 716\rangle \equiv
procedure ProcessSimpleTerm; virtual;
procedure StartFraenkelTerm; virtual;
procedure StartPostqualification; virtual;
procedure StartPostqualifyingSegment: virtual:
procedure ProcessPostqualifiedVariable; virtual;
procedure StartPostqualificationSpecyfication; virtual;
procedure FinishPostqualifyingSegment; virtual;
procedure FinishFraenkelTerm: virtual:
procedure StartSimpleFraenkelTerm; virtual;
procedure FinishSimpleFraenkelTerm; virtual;
procedure ProcessThesis; virtual;
procedure StartPrivateTerm; virtual;
procedure FinishPrivateTerm; virtual;
procedure StartBracketedTerm; virtual;
procedure FinishBracketedTerm; virtual;
procedure StartAggregateTerm; virtual;
procedure FinishAggregateTerm; virtual;
procedure StartSelectorTerm; virtual;
procedure FinishSelectorTerm; virtual;
procedure StartForgetfulTerm; virtual;
procedure FinishForgetfulTerm; virtual;
procedure StartChoiceTerm; virtual;
procedure FinishChoiceTerm; virtual;
procedure ProcessNumeralTerm; virtual;
procedure ProcessItTerm: virtual:
procedure ProcessLocusTerm; virtual;
procedure ProcessQua; virtual;
procedure FinishQualifiedTerm; virtual;
procedure ProcessExactly; virtual;
procedure StartLongTerm; virtual;
procedure ProcessFunctorSymbol: virtual:
procedure FinishArgList; virtual;
procedure FinishLongTerm; virtual;
procedure FinishArgument; virtual;
procedure FinishTerm; virtual;
procedure StartType: virtual:
procedure ProcessModeSymbol: virtual:
procedure Finish Type; virtual;
procedure Complete Type; virtual;
procedure ProcessAtomicFormula; virtual;
procedure ProcessPredicateSymbol; virtual;
procedure ProcessRightSideOfPredicateSymbol; virtual;
procedure FinishPredicativeFormula; virtual;
procedure FinishRightSideOfPredicativeFormula; virtual;
procedure StartMultiPredicativeFormula; virtual;
procedure FinishMultiPredicativeFormula; virtual;
procedure StartPrivateFormula; virtual;
procedure FinishPrivateFormula; virtual;
procedure ProcessContradiction; virtual;
procedure ProcessNegative; virtual;
```

226 SUBEXPRESSIONS Mizar Parser $\S716$

```
{ This is a temporary solution, the generation of ExpNodes is such that it is not possible to handle
      negation uniformly. }
    { Jest to tymczasowe rozwiazanie, generowanie ExpNode'ow jest takie, ze nie ma mozliwości obsluzenia
      jednolicie negacji. }
procedure ProcessNegation; virtual;
procedure FinishQualifyingFormula; virtual;
procedure FinishAttributiveFormula; virtual;
procedure ProcessBinaryConnective; virtual;
                                                \{+\}
procedure ProcessFlexDisjunction; virtual;
procedure ProcessFlexConjunction; virtual;
procedure StartRestriction; virtual;
procedure FinishRestriction; virtual;
procedure FinishBinaryFormula; virtual;
procedure FinishFlexDisjunction; virtual;
procedure FinishFlexConjunction; virtual;
procedure StartExistential; virtual;
procedure FinishExistential; virtual;
procedure StartUniversal; virtual;
procedure FinishUniversal; virtual;
procedure StartQualifiedSegment; virtual;
procedure StartQualifyingType; virtual;
procedure FinishQualifiedSegment; virtual;
procedure ProcessVariable; virtual;
procedure StartAttributes; virtual;
procedure ProcessNon; virtual;
procedure ProcessAttribute; virtual;
                                       \{+\}
procedure StartAttributeArguments; virtual;
procedure Complete Attribute Arguments; virtual; \{+\}
procedure FinishAttributeArguments; virtual;
                                                 \{+\}
procedure CompleteAdjectiveCluster; virtual;
procedure CompleteClusterTerm; virtual;
    { Errors Recovery}
procedure InsertIncorrTerm; virtual;
procedure InsertIncorrType; virtual;
procedure InsertIncorrBasic; virtual;
procedure InsertIncorrFormula; virtual;
See also section 1387.
This code is used in sections 717 and 1388.
```

§717 Mizar Parser SUBEXPRESSIONS 227

```
717.
       \langle Empty method declarations for SubexpObj 717\rangle \equiv
  (Methods implemented by subclasses of SubexpObj 716)
procedure FinishSample; virtual;
procedure ProcessThe; virtual;
procedure StartArgument; virtual;
procedure ProcessLeftParenthesis; virtual;
procedure ProcessRightParenthesis; virtual;
procedure StartAtomicFormula; virtual;
procedure ProcessHolds; virtual;
procedure FinishQuantified; virtual;
procedure ProcessNot; virtual;
procedure ProcessDoesNot; virtual;
procedure StartAdjectiveCluster; virtual;
procedure FinishAdjectiveCluster; virtual;
procedure FinishAttributes; virtual;
procedure CompleteAttributes; virtual;
procedure CompleteClusterType; virtual;
procedure FinishEquality; virtual;
```

This code is used in section 713.

228 SUBEXPRESSIONS Mizar Parser $\S718$

718.

```
\langle Subexpression procedures 718\rangle \equiv
procedure SubexpObj.StartAttributes;
  begin end;
procedure SubexpObj.StartAdjectiveCluster;
  begin end;
procedure SubexpObj.FinishAdjectiveCluster;
  begin end;
procedure SubexpObj.ProcessNon;
  begin end;
procedure SubexpObj.ProcessAttribute;
  begin end;
procedure SubexpObj.FinishAttributes;
  begin end;
procedure SubexpObj.CompleteAttributes;
  begin end;
procedure SubexpObj.StartAttributeArguments;
  begin end;
procedure SubexpObj.CompleteAttributeArguments;
  begin end;
procedure SubexpObj.FinishAttributeArguments;
  begin end;
procedure SubexpObj.CompleteAdjectiveCluster;
  begin end;
procedure SubexpObj.CompleteClusterTerm;
  begin end;
procedure SubexpObj.CompleteClusterType;
  begin end;
procedure SubexpObj.ProcessSimpleTerm;
  begin end;
procedure SubexpObj.ProcessQua;
  begin end;
procedure SubexpObj.FinishQualifiedTerm;
  begin end;
procedure SubexpObj.ProcessExactly;
  begin end;
procedure SubexpObj.StartArgument;
  begin end;
procedure SubexpObj.FinishArgument;
  begin end;
procedure SubexpObj.FinishTerm;
  begin end;
procedure SubexpObj.StartType;
  begin end;
procedure SubexpObj.ProcessModeSymbol;
  begin end;
procedure SubexpObj.FinishType;
  begin end;
procedure SubexpObj.CompleteType;
  begin end;
procedure SubexpObj.StartLongTerm;
  begin end;
```

```
procedure SubexpObj.FinishLongTerm;
  begin end;
procedure SubexpObj.FinishArgList;
  begin end;
procedure SubexpObj.ProcessFunctorSymbol;
  begin end;
procedure SubexpObj.StartFraenkelTerm;
  begin end;
procedure SubexpObj.FinishSample;
  begin end;
procedure SubexpObj.StartPostqualification;
  begin end;
procedure SubexpObj.StartPostqualificationSpecyfication;
  begin end:
procedure SubexpObj.StartPostqualifyingSegment;
  begin end;
procedure SubexpObj.ProcessPostqualifiedVariable;
  begin end;
procedure SubexpObj.FinishPostqualifyingSegment;
  begin end;
procedure SubexpObj.FinishFraenkelTerm;
  begin end;
procedure SubexpObj.StartSimpleFraenkelTerm;
  begin end;
procedure SubexpObj.FinishSimpleFraenkelTerm;
  begin end;
procedure SubexpObj.StartPrivateTerm;
  begin end;
procedure SubexpObj.FinishPrivateTerm;
  begin end;
procedure SubexpObj.StartBracketedTerm;
  begin end;
procedure SubexpObj.FinishBracketedTerm;
  begin end;
procedure SubexpObj.StartAggregateTerm;
  begin end;
procedure SubexpObj.FinishAggregateTerm;
  begin end;
{\bf procedure} \ \textit{SubexpObj.ProcessThe};
  begin end;
procedure SubexpObj.StartSelectorTerm;
  begin end;
{\bf procedure}\ {\it SubexpObj.FinishSelectorTerm};
  begin end;
procedure SubexpObj.StartForgetfulTerm;
  begin end;
procedure SubexpObj.FinishForgetfulTerm;
  begin end:
procedure SubexpObj.StartChoiceTerm;
  begin end;
procedure SubexpObj.FinishChoiceTerm;
  begin end;
```

```
procedure SubexpObj.ProcessNumeralTerm;
  begin end;
procedure SubexpObj.ProcessItTerm;
  begin end;
procedure SubexpObj.ProcessLocusTerm;
  begin end;
procedure SubexpObj.ProcessThesis;
  begin end;
{\bf procedure}\ {\it SubexpObj.StartAtomicFormula};
  begin end;
procedure SubexpObj.ProcessAtomicFormula;
  begin end;
procedure SubexpObj.ProcessPredicateSymbol;
  begin end:
procedure SubexpObj.ProcessRightSideOfPredicateSymbol;
  begin end;
procedure SubexpObj.FinishPredicativeFormula;
  begin end;
procedure SubexpObj.FinishRightSideOfPredicativeFormula;
  begin end;
procedure SubexpObj.StartMultiPredicativeFormula;
  begin end;
procedure SubexpObj.FinishMultiPredicativeFormula;
  begin end;
procedure SubexpObj.FinishQualifyingFormula;
  begin end;
procedure SubexpObj.FinishAttributiveFormula;
  begin end;
procedure SubexpObj.StartPrivateFormula;
  begin end;
procedure SubexpObj.FinishPrivateFormula;
  begin end;
procedure SubexpObj.ProcessContradiction;
  begin end;
procedure SubexpObj.ProcessNot;
  begin end;
procedure SubexpObj.ProcessDoesNot;
  begin end;
procedure SubexpObj.ProcessNegative;
  begin end;
procedure SubexpObj.ProcessNegation;
  begin end;
procedure SubexpObj.StartRestriction;
  begin end;
procedure SubexpObj.FinishRestriction;
  begin end;
procedure SubexpObj.ProcessHolds;
  begin end:
procedure SubexpObj.ProcessBinaryConnective;
  begin end;
procedure SubexpObj.FinishBinaryFormula;
  begin end;
```

```
procedure SubexpObj.ProcessFlexDisjunction;
  begin end;
procedure SubexpObj.ProcessFlexConjunction;
  begin end;
procedure SubexpObj.FinishFlexDisjunction;
  begin end;
procedure SubexpObj.FinishFlexConjunction;
  begin end;
procedure SubexpObj.StartQualifiedSegment;
  begin end;
procedure SubexpObj.StartQualifyingType;
  begin end;
procedure SubexpObj.FinishQualifiedSegment;
  begin end:
procedure SubexpObj.FinishQuantified;
  begin end;
{\bf procedure}\ {\it SubexpObj.ProcessVariable};
  begin end;
procedure SubexpObj.StartExistential;
  begin end;
procedure SubexpObj.FinishExistential;
  begin end;
procedure SubexpObj.StartUniversal;
  begin end;
procedure SubexpObj.FinishUniversal;
  begin end;
procedure SubexpObj.ProcessLeftParenthesis;
  begin end;
procedure SubexpObj.ProcessRightParenthesis;
  begin end;
{\bf procedure}\ {\it SubexpObj.} InsertIncorrType;
  begin end;
procedure SubexpObj.InsertIncorrTerm;
  begin end;
procedure SubexpObj.InsertIncorrBasic;
  begin end;
procedure SubexpObj.InsertIncorrFormula;
  begin end;
procedure SubexpObj.FinishEquality;
  begin end;
This code is used in section 683.
```

232 MSCANNER Mizar Parser §719

File 19

MScanner

```
719. We have the MScanner module transform an article (an input file) into a stream of tokens.
\langle \text{ scanner.pas } 588 \rangle + \equiv
         ⟨GNU License 4⟩
unit mscanner;
         interface
         uses errhan, mobjects, scanner;
                  ⟨ Public interface for MScanner 720⟩
         implementation
         uses mizenv;
                  ⟨Implementation for MScanner 726⟩;
         end.
720. Public types. We have enumerated types for each construction we'll encounter in Mizar.
\langle \text{ Public interface for MScanner } 720 \rangle \equiv
type (Token kinds for MScanner 724);
         CorrectnessKind = (syCorrectness, syCoherence, syCompatibility, syConsistency, syExistence,
                            syUniqueness, syReducibility);
         PropertyKind = (sErrProperty, sySymmetry, syReflexivity, syIrreflexivity, syAssociativity, syTransitivity, s
                            syCommutativity, syConnectedness, syAsymmetry, syIdempotence, syInvolutiveness, syProjectivity,
                            sySethood, syAbstractness);
         LibraryReferenceKind = (syThe, syDef, sySch);
         Directive Kind = (sy Vocabularies, sy Notations, sy Definitions, sy Theorems, sy Schemes, sy Registrations, sy Theorems, sy Theorems,
                            syConstructors, syRequirements, syEqualities, syExpansions);
         \langle Token type for MScanner 721\rangle;
See also sections 722 and 723.
This code is used in section 719.
721. Token type for MScanner.
\langle Token type for MScanner 721\rangle \equiv
          Token = \mathbf{record} \ Kind: \ TokenKind;
                            Nr: integer;
                            Spelling: string;
This code is used in section 720.
```

§722 Mizar Parser MSCANNER 233

722. Constants for MScanner

```
\langle \text{ Public interface for MScanner } 720 \rangle + \equiv
                { Homonymic and special symbols in buildin vocabulery }
        { Homonymic Selector Symbol }
    StrictSym = 1; \{ \text{"strict"} \}
         { Homonymic Mode Symbol }
    SetSym = 1; \{ \text{`set'} \}
        { Homonymic Predicate Symbol }
    EqualitySym = 1; \{ '=' \}
        { Homonymic Circumfix Symbols }
    SquareBracket = 1; \{ `[` `]` \}
    CurlyBracket = 2; \{ "-" "" \}
    RoundedBracket = 3; \{ "("")" \}
    scTooLongLineErrorNr = 200;  { Error number: Too long line }
    ⟨ Token names for MScanner 725⟩;
CorrectnessName: array [CorrectnessKind] of string = ('correctness', 'coherence',
                  'compatibility', 'consistency', 'existence', 'uniqueness', 'reducibility');
PropertyName: array [PropertyKind] of string = (``, `symmetry', `reflexivity', `irreflexivity', `array [PropertyKind] of string = (``, `symmetry', `reflexivity', `irreflexivity', `irreflexivi
                 'associativity', 'transitivity', 'commutativity', 'connectedness', 'asymmetry',
                 'idempotence', 'involutiveness', 'projectivity', 'sethood', 'abstractness');
LibraryReferenceName: array [LibraryReferenceKind] of string = ('the', 'def', 'sch');
DirectiveName: array [DirectiveKind] of
        string = ('vocabularies', 'notations', 'definitions', 'theorems', 'schemes', 'registrations',
                  'constructors', 'requirements', 'equalities', 'expansions');
PlaceHolderName: array [1..10] of
        string = (`$1`, `$2`, `$3`, `$4`, `$5`, `$6`, `$7`, `$8`, `$9`, `$10`);
    Unexpected = sErrProperty;
723. Public facing procedures and global variables. Of particular importance, the global variable
gScanner is declared here.
\langle \text{Public interface for MScanner } 720 \rangle + \equiv
var PrevWord, CurWord, AheadWord: Token;
    PrevPos, AheadPos: Position;
procedure ReadToken;
procedure LoadPrf(const aPrfFileName: string);
procedure DisposePrf;
procedure StartScaner;
procedure InitSourceFile(const aFileName, aDctFileName: string);
procedure CloseSourceFile;
procedure InitScanning(const aFileName, aDctFileName: string);
procedure FinishScanning:
var gScanner: MScannPtr = nil; { This is important }
    ModeMaxArgs, StructModeMaxArgs, PredMaxArgs: IntSequence;
```

234 MSCANNER Mizar Parser $\S724$

724. Token kinds. If I were cleverer, I would have some WEB macros to make this readable.

```
\langle Token kinds for MScanner 724\rangle \equiv
  TokenKind = (syT\theta, \{ \#0 \})
     syT1, { #1 }
     syT2,
                 #2
     syT3,
                 #3
     syT4,
                 #4
     syT5,
                 #5
     syT6,
                 #6
     syT7,
                 #7
     syT8,
                 #8
                 #9 }
     syT9,
     syT10,
               { #10
     syT11,
                  #11
     syT12,
                  #12
     syT13,
                  #13
     syT14,
                  #14
     syT15,
                  #15
     syT16,
                  #16
     syT17,
                  #17
     syT18,
                  #18
     syT19,
                  #19
     syT20,
                  #20
     syT21,
                  #21
     syT22,
                  #22
     syT23,
                  #23
     syT24,
                  #24
     syT25,
                  #25
     syT26,
                  #26
     syT27,
                  #27
     syT28,
                  #28
     syT29,
                  #29
     syT30,
                  #30
               { #31
     syT31,
     Pragma, { #32 }
     EOT = 33, \{ ! \#33 \}
     sy\_from, \ \{"\#34 \} \\ sy\_identify, \ \{\#\#35 \} 
     sy_thesis, { $ #36 }
     sy\_contradiction, { % #37 }
     sy\_Ampersand, { & #38 }
     sy_{-}by, { ' #39 }
     sy\_LeftParanthesis, { ( #40 }
     sy_RightParanthesis, { ) #41 }
     sy\_registration, {* #42 }
     sy\_definition, {+ #43 }
     sy\_Comma, { , #44 }
     \begin{array}{lll} sy\_notation\,, & \{\,-\, \, \#45\, \,\,\} \\ sy\_Ellipsis\,, & \{\,.\, \,\, \#46\, \,\,\} \end{array}
     sy_proof, { / #47 }
     syT48, { 0 #48 }
     syT49, {1 #49 }
```

 $\S724$ Mizar Parser MSCANNER 235

```
syT50, {2 #50 }
syT51, {3 #51 }
syT52, {4 #52 }
syT53, { 5 #53
syT54, {6 #54
syT55, {7 #55
syT56, {8 #56 }
syT57, { 9 #57 }
sy_{-}Colon, {: #58 }
sy\_Semicolon, {; #59}
sy_now, {< #60 }
sy\_Equal, {= #61 }
sy_{-}end, { > #62 }
sy\_Error, {? #63}
syT64, { @ #64 }
MMLIdentifier, \{A \#65 \}
syT66, {B #66}
syT67, {C #67 }
sy_LibraryDirective, {D #68 } {see DirectiveKind}
syT69, {E #69 }
syT70, {F #70 }
StructureSymbol, {G #71 }
syT72, {H #72}
Identifier, \{I \#73 \}
ForgetfulFunctor, {J #74 }
LeftCircumfixSymbol, { K #75 }
RightCircumfixSymbol, {L #76 }
ModeSymbol, {M #77 }
Numeral, {N #78}
InfixOperatorSymbol, { 0 #79 }
syT80, {P #80 }
ReferenceSort, \{Q \#81 \}
PredicateSymbol, {R #82 }
syT83, {S #83 }
syT84, {T #84}
SelectorSymbol, {U #85 }
AttributeSymbol, { V #86 }
syT87, {W #87}
sy_Property, {X #88 } { see PropertyKind }
sy\_CorrectnessCondition, {Y #89 } {see CorrectnessKind}
sy\_Dolar, {Z #90 } { $1 $2 $3 $4 $5 $6 $7 $8 $9 $10 }
sy_LeftSquareBracket, { [ #91 }
syT92, { #92 }
sy_RightSquareBracket, {] #93 }
syT94, { ^* #94 }
syT95, { _{-} #95 }
syT96, { ' #96 }
sy\_according, {a #97}
syT98, {b #98 }
sy\_reduce, { c #99 }
syT100, {d #100}
sy\_equals, {e #101}
```

Mizar Parser

§724

```
syT102, {f #102}
syT103, {g #103}
sy_with, {h #104}
syT105, {i #105}
syT106, {j #106}
syT107, {k #107}
syT108, {1 #108}
syT109, {m #109}
syT110, {n #110}
syT111, { o #111 }
syT112, {p #112}
syT113, {q #113}
sy_{-}wrt = 114, {r #114}
syT115, {s #115}
sy_to, {t #116}
syT117, {u #117}
syT118, {v #118}
sy\_when, {w #119}
sy\_axiom, {x #120}
syT121, {y #121}
syT122, {z #122}
sy\_LeftCurlyBracket, { \#123 }
syT124, {| #124}
sy_RightCurlyBracket, { #125}
syT126, {~ #126}
syT127, { #127 }
syT128, { #128 }
syT129, { #129 }
syT130, { #130 }
syT131, { #131 }
syT132,
        { #132 }
syT133, { #133 }
syT134, { #134 }
sy\_correctness = 135, \quad \{ \#135 \}
syT136, { #136 }
syT137, { #137 }
syT138, { #138 }
syT139, { #139 }
sy_{-}if = 140, \{ \#140 \}
syT141, { #141 }
syT142, { #142 }
syT143, {#143}
sy_i s = 144, \{ \#144 \}
sy\_are, { #145 }
syT146, { #146 }
sy\_otherwise, { #147 }
syT148, { #148 }
syT149, { #149 }
syT150, { #150 }
syT151, {#151}
syT152, { #152 }
syT153, { #153 }
```

```
syT154, { #154 }
syT155, { #155 }
sy_{-}ex = 156, \{ #156 \}
sy\_for, { #157 }
syT158, { #158 }
sy_define, { #159 }
syT160, {#160}
sy\_being, { #161 }
sy\_over, { #162 }
syT163, { #163 }
sy\_canceled, { #164 }
sy_{-}do, { #165 }
sy_does, { #166 }
sy_{-}or, { #167 }
sy_where, { #168 }
sy_non, { #169 }
sy\_not, { #170 }
sy\_cluster, { #171 }
sy_-attr, { #172 }
syT173, {#173}
sy\_StructLeftBracket, { #174 }
sy\_StructRightBracket, { #175 }
sy_environ, { #176 }
syT177, { #177 }
sy\_begin, { #178 }
syT179, { #179 }
syT180, {#180}
syT181, { #181 }
syT182, { #182 }
syT183, { #183 }
syT184, { #184 }
sy_hence, { #185 }
syT186, { #186 }
syT187, { #187 }
sy\_hereby, { #188 }
syT189, { #189 }
syT190, { #190 }
syT191, { #191 }
sy\_then, { #192 }
sy\_DotEquals, { #193 }
syT194, { #194 }
syT195, { #195 }
sy\_synonym, { #196 }
sy\_antonym, { #197 }
syT198, { #198 }
syT199, {#199}
sy_let, { #200 }
sy_{-}take, \{ #201 \}
sy_assume, { #202 }
sy\_thus, \quad \{ \, \texttt{#203} \, \}
sy\_given, { #204 }
sy\_suppose, { #205 }
```

238 MSCANNER

```
sy\_consider, { #206 }
syT207, { #207 }
syT208, { #208 }
syT209, { #209 }
syT210, { #210 }
sy-Arrow, { #211 }
sy_as, { #212 }
sy\_qua, { #213 }
sy_be, { #214 }
sy\_reserve, { #215 }
syT216, { #216 }
syT217, { #217 }
syT218, { #218 }
syT219, { #219 }
syT220, { #220 }
syT221, { #221 }
syT222, { #222 }
syT223, { #223 }
sy_set, { #224 }
sy\_selector, { #225 }
sy\_cases, { #226 }
sy_per, \{ #227 \}
sy\_scheme, {#228}
sy\_redefine, {#229}
sy\_reconsider, { #230 }
sy\_case, { #231 }
sy\_prefix, { #232 }
sy\_the, { #233 }
sy_it, { #234 }
sy_{-}all, { #235 }
sy\_theorem, { #236 }
sy\_struct, { #237 }
sy\_exactly, { #238 }
sy\_mode, { #239 }
sy_-iff, { #240 }
sy_func, { #241 }
sy\_pred, { #242 }
sy_implies, { #243 }
sy_-st, { #244 }
sy\_holds, { #245 }
sy\_provided, { #246 }
sy_means, { #247 }
sy_{-}of, { #248 }
sy\_defpred, { #249 }
sy\_deffunc, { #250 }
sy\_such, { #251 }
sy_-that, { #252 }
sy\_aggregate, { #253 }
sy_{-}and = \{ \#254 \});
```

This code is used in section 720.

§725 Mizar Parser MSCANNER 239

725. We have string representation for each of the token kinds, which is useful for debugging purposes.

```
\langle Token names for MScanner 725\rangle \equiv
TokenName: array [TokenKind] of string = (``, {#0})
    '', { #1 }
'', { #2 }
'', { #3 }
''', { #4 }
            #4
            #5
            #6
            #7
            #8
            #9
            #10 }
            #11
            #12
            #13
            #14
            #15
            #16
            #17
            #18
            #19
            #20
            #21
            #22
            #23
            #24
            #25
            #26
            #27
            #28
            #29
            #30
            #31
         { #32 }
    ~~, {! #33 }
    from', { " #34 }
identify', { # #35 }
    'thesis', { $ #36 }
     'contradiction', {% #37 }
     '&', {& #38 }
    'by', {' #39 }
    '(', { ( #40 }
     ')', {) #41 }
     'registration', {* #42 }
    'definition', {+ #43 }
     `, `, { , #44 }
    notation', {- #45 }
...', {. #46 }
     'proof', {/ #47 }
    ·, {0 #48 }
    ···, {1 #49 }
```

240 MSCANNER Mizar Parser §725

```
17, {2 #50 }
17, {3 #51 }
···, {4 #52 }
7, {5 #53 }
7, {6 #54 }
7, {7 #55 }
7, {8 #56 }
´´, {9 #57 }
':', {: #58 }
';', {; #59 }
'now', {< #60 }
'=', {= #61 }
'end', {> #62 }
? #63 }

7, {? #63 }

7, {0 #64 }

7, {A #65 }

7, {B #66 }

7, {C #67 }
'vocabularies', {D #68 }
'', {E #69 }
7, {F #70 }
7, {G #71 }
7, {H #72 }
7, {T #72 }
    {I #73 }
; {J #74 }
  , {K #75 }
   {L #76 }
'
', {L #76 }
', {M #77 }
', {N #78 }

;, {N #78 }
77, {0 #79 }
77, {P #80 }
'def', {Q #81 }
~~, {R #82 }
'`, {S #83 }
'`, {T #84 }
'`, {U #85 }
··, {V #86 }
´´, {W #87 }
'symmetry', { X #88 }
'coherence', {Y #89 }
`$1``, {Z #90 }
`[`, {[ #91 }
··, {'⊔' #92 }
´]´, {] #93 }
( #94 )

', { #95 }
', { #96 }

'according', \{a \# 97 \}
'`, {b #98 }
'reduce', {c #99 }
", {d #100}
'equals', {e #101}
```

§725 Mizar Parser

```
``, {f #102}
``, {g #103}
'with', {h #104}
``, \{i \#105\}
7, {1 #105}
7, {j #106}
7, {k #107}
7, {l #108}
7, {m #109}
7, {n #110}
7, {o #111}
7, {o #111}
7, {p #112}
7, {q #113}
'wrt', {r #114}
´´, {s #115}
'to', {t #116}

'( u #117)
'( v #118)
'()
'when', {w #119}
'axiom', {x #120}
'', {y #121}
'', {z #122}
'{', { #123}
( #124 )
'}', { #125}
'', { #126}
'T127', {#127}
``, {#128}
'T129', {#129}
``, {#130}
T131', {#131}
7, {#132}
7, {#133}
7, {#134}
'correctness', \{ #135 \}
'T136', {#136}
``, {#137}
7, {#138}
7, {#139}
7if, {#140}
(#141)
(#142)
(#143)
is', {#144}
fare, { #145 }
``, {#146}
'otherwise', \{ #147 \}
7, {#148}
7, {#149}
7, {#150}
7, {#151}
'T152', {#152}
11, { #153 }
```

242 MSCANNER Mizar Parser $\S725$

```
'`, {#154}
'`, {#155}
 'ex', {#156}
for', {#157}
 ``, {#158}
 'define', { #159 }
 11, { #160 }
 'being', {#161}
 foverf, \{ #162 \}
 ´´, {#163}
 'canceled', { #164 }
 'do', {#165}
 'does', { #166 }
 for, { #167 }
 'where', {#168}
 'non', {#169}
'not', {#170}
 'cluster', {#171}
 'attr', {#172}
 ``, { #173 }
 (\#', {#174}
 ^\#) ^, { #175 }
 'environ', {#176}
 ··, {#177}
 'begin', {#178}
 ``, {#179}
; {#180}; ; {#181}; ; {#182}; ; {#183}; ; {#184};
 'hence', { #185 }
 ``, {#186}
··, {#187}
 'hereby', { #188 }
 ``, {#189}
; {#190}; {#191}
 'then', \{ #192 \}
 1.=1, {#193}
         , {#194}
 ´´, {#195}
 'synonym', { #196 }
 fantonym fantony

'', {#198}
'', {#199}
 'let', {#200}
 \texttt{`take'}, \quad \{\, \texttt{#201}\, \}
 assume, { #202 }
 'thus', {#203}
 'given', {#204}
 'suppose', {#205}
```

```
'consider', { #206 }
··, {#207}
; {#208}
;; {#209}
;; {#210}
`->`, {#211}
`as`, {#212}
'qua', {#213}
'be', {#214}
reserve', {#215}
;; {#216}
, {#210}
, {#217}
, {#218}
, {#219}
, {#220}
, {#221}
, {#222}
  ´, {#222 }
~~, {#223}
'set', {#224}
selector', \{#225\}
'cases', { #226 }
'per', {#227}
'scheme', \{ \#228 \} redefine', \{ \#229 \}
'reconsider', \{ #230 \}
'case', {#231}
'prefix', { #232 }
'the', \{ #233 \}
'it', {#234}
'all', {#235}
'theorem', \{ \#236 \}
'struct', {#237}
'exactly', { #238 }
'mode', {#239}
iff', {#240}
funcf, \{ #241 \}
'pred', { #242 }
'implies', {#243}
'st', {#244}
'holds', { #245 }
'provided', { #246 }
'means', {#247}
'of', {#248}
'defpred', \{ #249 \} 'deffunc', \{ #250 \}
'such', {#251}
'that', { #252 }
'aggregate', {#253}
'and' {#254})
```

This code is used in section 722.

244 MSCANNER Mizar Parser §726

726. Reading a token. This tokenizes a Mizar article, using the scanner's GetToken method. We can trace this GetToken back to its implementation (§629). This, in turn, depends on the SliceIt method (§612). This method is used to determine the next token in parser.pas's Parse function.

This assumes that StartScanner (§729) has been invoked already, which initializes the CurWord token and other variables.

Also important to observe: the Kind of the token is populated here. $\langle \text{Implementation for MScanner 726} \rangle \equiv$ **procedure** ReadToken; **begin** $PrevWord \leftarrow CurWord$; $PrevPos \leftarrow CurPos$; $CurWord \leftarrow AheadWord$; $CurPos \leftarrow AheadPos$; $\{ \ '\cdot '$ is not allowed in an identifiers in the text proper $\}$ **if** $(CurWord.Kind = sy_Begin)$ **then** $gScanner \uparrow .Allowed [\ ' _ '] \leftarrow 0$; **if** $(CurWord.Kind = sy_Error) \land (CurWord.Nr = scTooLongLineErrorNr)$ **then** ErrImm(CurWord.Nr); $gScanner \uparrow .GetToken$; $AheadWord.Kind \leftarrow TokenKind(gScanner \uparrow .fLexem.Kind)$; $AheadWord.Nr \leftarrow gScanner \uparrow .fLexem.Nr$; $AheadWord.Spelling \leftarrow gScanner \uparrow .fStr$; $AheadPos \leftarrow gScanner \uparrow .fPos$; **end**;

See also sections 727, 728, 729, 730, 731, 732, and 733.

This code is used in section 719.

727. Loading a proof file. The .prf file is a file containing numerals, and its usage eludes me. The format consists of multiple lines:

Line 1: Three non-negative integers are on the first line "M S P"

Line 2: Contains M non-negative integers separated by a single whitespace

Line 3: Contains S non-negative integers separated by a single whitespace

Line 4: Contains P non-negative integers separated by a single whitespace.

This function loads the contents of the .prf file. This initializes the global variables ModeMaxArgs, StructureModeMaxArgs, PredMaxArgs, then populates them.

```
\langle Implementation for MScanner 726\rangle + \equiv
procedure LoadPrf (const aPrfFileName: string);
  var lPrf: text; lModeMaxArgsSize, lStructModeMaxArgsSize, lPredMaxArgsSize, i, lInt, r: integer;
  begin assign(lPrf, aPrfFileName + `.prf'); reset(lPrf);
  Read(lPrf, lModeMaxArqsSize, lStructModeMaxArqsSize, lPredMaxArqsSize);
  ModeMaxArgs.Init(lModeMaxArgsSize + 1); r \leftarrow ModeMaxArgs.Insert(0);
  StructModeMaxArgs.Init(lStructModeMaxArgsSize + 1); r \leftarrow StructModeMaxArgs.Insert(0);
  PredMaxArgs.Init(lPredMaxArgsSize + 1); r \leftarrow PredMaxArgs.Insert(0);
  for i \leftarrow 1 to lModeMaxArgsSize do
    begin Read(lPrf, lInt); r \leftarrow ModeMaxArgs.Insert(lInt);
    end:
  for i \leftarrow 1 to lStructModeMaxArqsSize do
    begin Read(lPrf, lInt); r \leftarrow StructModeMaxArgs.Insert(lInt);
  for i \leftarrow 1 to lPredMaxArgsSize do
    begin Read(lPrf, lInt); r \leftarrow PredMaxArgs.Insert(lInt);
    end;
  close(lPrf);
  end;
```

§728 Mizar Parser MSCANNER 245

```
We cleanup after using the .prf file.
\langle Implementation for MScanner 726\rangle + \equiv
procedure DisposePrf;
  begin ModeMaxArqs.Done; PredMaxArqs.Done; StructModeMaxArqs.Done;
  end:
729. We construct an MScann object to scan a file.
\langle Implementation for MScanner 726\rangle + \equiv
procedure StartScaner;
  begin CurPos.Line \leftarrow 1; CurPos.Col \leftarrow 0; AheadWord.Kind \leftarrow TokenKind(qScanner \uparrow fLexem.Kind);
  AheadWord.Nr \leftarrow gScanner \uparrow .fLexem.Nr; AheadWord.Spelling \leftarrow gScanner \uparrow .fStr;
  AheadPos \leftarrow gScanner \uparrow .fPos;
  end:
730. We initialize a scanner for a file.
\langle Implementation for MScanner 726\rangle + \equiv
procedure InitSourceFile(const aFileName, aDctFileName: string);
  begin new(gScanner, InitScanning(aFileName, aDctFileName)); StartScaner;
  end;
       When we're done with a scanner, we call the destructor for the MScanner.
\langle Implementation for MScanner 726\rangle + \equiv
procedure CloseSourceFile;
  begin dispose(gScanner, Done);
  end:
732. We can combine the previous functions together to initialize a scanner for a file (an article) and its
dictionary file.
\langle Implementation for MScanner 726\rangle + \equiv
procedure InitScanning(const aFileName, aDctFileName: string);
  begin gScanner \leftarrow new(MScannPtr, InitScanning(aFileName, aDctFileName)); StartScaner;
  LoadPrf(aDctFileName);
  end;
       We cleanup after scanning, saving a dictionary XML file to an ".idx" file. This uses the global
variable EnvFileName declared in mizenv.pas (§24).
\langle Implementation for MScanner 726\rangle + \equiv
procedure FinishScanning;
  begin gScanner↑.fIdents.SaveXDct(EnvFileName + ´.idx´); CloseSourceFile; DisposePrf;
  end:
```

246 ABSTRACT SYNTAX Mizar Parser §734

File 20

Abstract Syntax

734. A crucial step in any interpreter, compiler, or proof assistant is to transform the concrete syntax into an abstract syntax tree. This module provides all the classes for the abstract syntax tree of expressions, types, and formulas in Mizar. The abstract syntax tree for "statements" will be found in the "Weakly Strict Text Proper" module.

This is a bit, well, "Java-esque", in the sense that each different kind of node in the abstract syntax tree is represented by a different class. If you don't know abstract syntax trees, I can heartily recommend Bob Nystrom's *Crafting Interpreters* (Ch. 5: Representing Code) for an overview.

I'll be quoting from the grammar for Mizar as we go along, since the class hierarchy names their classes after the nonterminal symbols in the grammar. (It's what anyone would do.) You can find a local copy of the grammar on most UNIX machines with Mizar installed located at /usr/local/doc/Mizar/syntax.txt, which you can study at your leisure.

735. Warning: There is a lot of boiler plate code in the constructors and destructors. I am going to pass over them without much comment, because they are monotonous and uninteresting. The more interesting part will be discussed with the class declarations for each kind of node. I will simply entitle the paragraphs "Constructor" to indicate I am recognizing their existence and moving on.

```
 \langle \text{abstract\_syntax.pas 735} \rangle \equiv \\ \langle \text{GNU License 4} \rangle \\ \text{unit } abstract\_syntax; \\ \text{interface uses } errhan, mobjects, syntax; \\ \langle \text{Interface for abstract syntax 737} \rangle \\ \text{implementation} \\ \langle \text{Implementation of abstract syntax 736} \rangle \\ \text{end} \ .
```

736. The implementation requires discussing a few "special cases" (variables, qualified segments, adjectives) before getting to the usual syntactic classes (terms, types, formulas).

```
\label{eq:asymptotic_loss} $$ \langle \mbox{Implementation of abstract syntax 736} \rangle \equiv $$ \langle \mbox{Variable AST constructor 739} \rangle $$ \langle \mbox{Qualified segment AST constructor 742} \rangle $$ \langle \mbox{Adjective expression AST constructor 748} \rangle $$ \langle \mbox{Adjective AST constructor 752} \rangle $$ \langle \mbox{Negated adjective AST constructor 750} \rangle $$ \langle \mbox{Implementing term AST 757} \rangle $$ \langle \mbox{Implementing type AST 799} \rangle $$ \langle \mbox{Implementing formula AST 811} \rangle $$ \langle \mbox{Within expression AST implementation 1664} \rangle $$
```

This code is used in section 735.

 $\S737$ Mizar Parser ABSTRACT SYNTAX 247

737. The interface consists mostly of classes, as well as a few enumerated types. The gambit resembles what we would do if we were programming in C: define an enum TermSort, then introduce a struct TermAstNode {enum TermSort sort;} to act as an abstract base class for terms (and do likewise for formulas, types, etc.). This allows us to use "struct inheritance" in C, as Bob Nystrom's Crafting Interpreters (Ch. 19) calls it.

738. Variable. A variable in the abstract syntax tree is basically a de Bruijn index, in the sense that it is represented by an integer in the metalanguage (PASCAL).

Logicians may feel uncomfortable at variables being outside the term syntax tree. But what logicians think of as "variables" in first-order logic, Mizar calls them "Simple Terms" (§756).

```
⟨ Variable (abstract syntax tree) 738⟩ ≡
    VariablePtr = ↑VariableObj;
    VariableObj = object (MObject)
        nIdent: integer; { identifier number }
        nVarPos: Position;
        constructor Init(const aPos: Position; aIdentNr: integer);
    end
This code is used in section 737.

739. Constructor.
⟨ Variable AST constructor 739⟩ ≡
    constructor VariableObj.Init(const aPos: Position; aIdentNr: integer);
    begin nIdent ← aIdentNr; nVarPos ← aPos;
    end;
This code is used in section 736.
```

248 ABSTRACT SYNTAX Mizar Parser $\S740$

740. Qualified segment. A qualified segment refers to situations in, e.g., "consider $\langle qualified - segment \rangle^+$ such that ...". This also happens in quantifiers where the Working Mathematician writes $\forall \vec{x}. P[\vec{x}]$, for example (that quantifier prefix " $\forall \vec{x}$ " uses the qualifying segment \vec{x}).

The Mizar grammar for qualified segments looks like:

We will implement Qualified-Variables as an array of pointers to QualifiedSegment objects, each one being either implicit or explicit.

741. Abstract base class for qualified segments. We have *implicitly* qualified segments and *explicitly* qualified segments, which are "both" qualified segments. Object-oriented yoga teaches us to describe this situation using a "qualified segment" abstract base class, and then extend it with two subclasses.

```
\langle Qualified segment (abstract syntax tree) 741\rangle \equiv
  SegmentKind = (ikImplQualifiedSegm, ikExplQualifiedSegm);
  QualifiedSegmentPtr = \uparrow QualifiedSegmentObj;
  QualifiedSegmentObj = \mathbf{object} (MObject)
     nSegmPos: Position;
     nSegmentSort: SegmentKind;
     constructor Init(const aPos: Position; aSort: SegmentKind);
  end
See also sections 743 and 745.
This code is used in section 737.
       Constructor.
\langle \text{ Qualified segment AST constructor } 742 \rangle \equiv
constructor QualifiedSegmentObj.Init(const aPos: Position; aSort: SegmentKind);
  begin nSeqmPos \leftarrow aPos; nSeqmentSort \leftarrow aSort;
  end;
See also sections 744 and 746.
This code is used in section 736.
```

743. Implicitly qualified segments. When we use "reserved variables" in the qualifying segment, we can suppress the type ascription (i.e., the "being $\langle Type \rangle$ "). This makes the typing *implicit*. Hence the name *implicitly* qualified segments (the types are implicitly given).

```
⟨ Qualified segment (abstract syntax tree) 741⟩ +≡
ImplicitlyQualifiedSegmentPtr = ↑ImplicitlyQualifiedSegmentObj;
ImplicitlyQualifiedSegmentObj = object (QualifiedSegmentObj)
    nIdentifier: VariablePtr;
    constructor Init(const aPos: Position; aIdentifier: VariablePtr);
    destructor Done; virtual;
end;
```

 $\S744$ Mizar Parser ABSTRACT SYNTAX 249

```
Constructor. The constructors and destructors for implicitly qualified segments are straightforward.
\langle \text{Qualified segment AST constructor } 742 \rangle + \equiv
constructor Implicitly Qualified Segment Obj. Init (const a Pos: Position; a Identifier: Variable Ptr);
  begin inherited Init(aPos, ikImplQualifiedSeqm); nIdentifier \leftarrow aIdentifier;
  end:
destructor Implicitly Qualified Segment Obj. Done;
  begin dispose (nIdentifier, Done);
  end:
745. Explicitly qualified segment. The other possibility in Mizar is that we will have "explicitly typed
variables" in the qualifying segment. The idea is that, in Mizar, we can permit the following situation:
    consider x,y,z being set such that ...
This means the three variables x, y, z are explicitly qualified variables with the type "set". We represent
this using one Explicitly Qualified Segment object, a vector for the identifiers (x, y, z) and a pointer to their
type (set).
\langle \text{Qualified segment (abstract syntax tree) } 741 \rangle + \equiv
  Explicitly Qualified Segment Ptr = \uparrow Explicitly Qualified Segment Obj;
  Explicitly Qualified Segment Obj = \mathbf{object} (Qualified Segment Obj)
    nIdentifiers: PList; { of identifier numbers }
    nType: TypePtr;
    constructor Init(const aPos: Position; aIdentifiers: PList; aType: TypePtr);
    destructor Done; virtual;
  end
       The constructors and destructors for explicitly qualified segments are straightforward.
\langle Qualified segment AST constructor 742 \rangle + \equiv
{\bf constructor}\ {\it Explicitly Qualified Segment Obj. Init} ({\bf const}\ aPos:\ Position;
                                                    aIdentifiers: PList;
                                                     aType: TypePtr);
  begin inherited Init(aPos, ikExplQualifiedSeqm); nIdentifiers \leftarrow aIdentifiers; nType \leftarrow aType;
destructor Explicitly Qualified Segment Obj. Done;
  begin dispose(nIdentifiers, Done); dispose(nType, Done);
  end:
747.
       Attributes. Attributes can have arguments preceding it. The relevant part of the Mizar grammar,
I think, is:
        Adjective-Cluster = { Adjective } .
        Adjective = [ "non" ] [ Adjective-Arguments ] Attribute-Symbol .
\langle Adjective expression (abstract syntax tree) 747\rangle \equiv
  AdjectiveSort = (wsNegatedAdjective, wsAdjective);
  AdjectiveExpressionPtr = \uparrow AdjectiveExpressionObj;
  AdjectiveExpressionObj = \mathbf{object} (MObject)
    nAdjectivePos: Position;
    nAdjectiveSort: AdjectiveSort;
    constructor Init(const aPos: Position; aSort: AdjectiveSort);
    destructor Done; virtual;
This code is used in section 737.
```

250 ABSTRACT SYNTAX Mizar Parser $\S748$

```
\langle Adjective expression AST constructor 748 \rangle \equiv
constructor AdjectiveExpressionObj.Init(const aPos: Position; aSort: AdjectiveSort);
  begin nAdjectivePos \leftarrow aPos; nAdjectiveSort \leftarrow aSort;
  end:
destructor AdjectiveExpressionObj.Done;
  begin end;
This code is used in section 736.
749. Negated adjective. We represent an adjective using the EBNF grammar (c.f., the WSM article-
related function InWSMizFileObj.Read_Adjective:AdjectiveExpressionPtr):
        Negated-Adjective ::= "non" Adjective-Expr;
        Positive-Adjective ::= [Adjective-Arguments] Attribute-Symbol;
        Adjective-Expr ::= Negated-Adjective | Positive-Adjective;
Hence we only really need a pointer to the "adjective being negated".
\langle \text{Negated adjective expression (abstract syntax tree) } 749 \rangle \equiv
  NegatedAdjectivePtr = \uparrow NegatedAdjectiveObj;
  NegatedAdjectiveObj = \mathbf{object} \ (AdjectiveExpressionObj)
    nArg: AdjectiveExpressionPtr;  { of TermPtr, visible arguments }
    constructor Init(const aPos: Position; aArg: AdjectiveExpressionPtr);
    destructor Done; virtual;
  end
This code is used in section 737.
750. Constructor.
\langle Negated adjective AST constructor 750 \rangle \equiv
constructor NegatedAdjectiveObj.Init(const aPos: Position; aArg: AdjectiveExpressionPtr);
  begin inherited Init(aPos, wsNegatedAdjective); nArg \leftarrow aArg;
destructor NegatedAdjectiveObj.Done;
  begin dispose(nArg, Done);
  end:
This code is used in section 736.
751. Adjective objects. This is the preferred node for later intermediate representations for attributes,
since nNegated is a field in the class.
\langle \text{Adjective (abstract syntax tree) } 751 \rangle \equiv
  AdjectivePtr = \uparrow AdjectiveObj;
  AdjectiveObj = \mathbf{object} \ (AdjectiveExpressionObj)
    nAdjectiveSymbol: integer;
    nNegated: boolean;
    nArgs: PList; { of TermPtr, visible arguments }
    constructor Init(const aPos: Position; aAdjectiveNr: integer; aArgs: PList);
    destructor Done; virtual;
  end
This code is used in section 737.
```

 $\S752$ Mizar Parser ABSTRACT SYNTAX 251

752. Constructor.

```
 \langle \text{Adjective AST constructor 752} \rangle \equiv \\ \textbf{constructor } \textit{AdjectiveObj.Init}(\textbf{const } aPos: \textit{Position}; \textit{aAdjectiveNr}: \textit{integer}; \textit{aArgs}: \textit{PList}); \\ \textbf{begin } \textit{inheritedInit}(\textit{aPos}, \textit{wsAdjective}); \textit{nAdjectiveSymbol} \leftarrow \textit{aAdjectiveNr}; \textit{nArgs} \leftarrow \textit{aArgs}; \\ \textbf{end}; \\ \textbf{destructor } \textit{AdjectiveObj.Done}; \\ \textbf{begin } \textit{dispose}(\textit{nArgs}, \textit{Done}); \\ \textbf{end}; \\ \end{cases}
```

This code is used in section 736.

Section 20.1. TERMS (ABSTRACT SYNTAX TREE)

753. We have an abstract base class for terms, along with the "sorts" (syntactic subclasses) allowed. This allows, e.g., formulas, to refer to terms without knowing the sort of term involved. The UML class diagram for term:

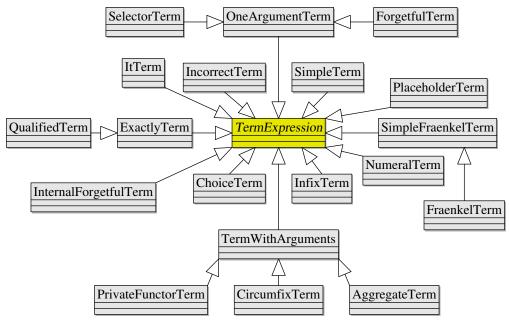


Fig. 3. UML class diagram for abstract syntax tree for terms.

The arrows indicate inheritance, pointing from the subclass to the parent superclass. The abstract base class TermExpression is italicized, but it is so difficult to distinguish we have colored it yellow.

NOTE: the class UML diagram may be missing a few descendents of *TermExpression*, but it contains the important subclasses which I could fit into it.

```
 \langle \text{Abstract base class for terms 753} \rangle \equiv \\ TermSort = (wsErrorTerm, wsPlaceholderTerm, wsNumeralTerm, wsSimpleTerm, \\ wsPrivateFunctorTerm, wsInfixTerm, wsCircumfixTerm, wsAggregateTerm, wsForgetfulFunctorTerm, \\ wsInternalForgetfulFunctorTerm, wsSelectorTerm, wsInternalSelectorTerm, wsQualificationTerm, \\ wsGlobalChoiceTerm, wsSimpleFraenkelTerm, wsFraenkelTerm, wsItTerm, wsExactlyTerm); \\ TermPtr = \uparrow TermExpressionObj; \\ TermExpressionObj = \mathbf{object} \ (MObject) \\ nTermSort: TermSort; \\ nTermPos: Position; \\ \mathbf{end} \\ \\ \text{This code is used in section 737}.
```

```
754.
     The grammar for term expressions in Mizar as stated in syntax.txt:
      Term-Expression = "(" Term-Expression ")"
         [ Arguments ] Functor-Symbol [ Arguments ]
          Left-Functor-Bracket Term-Expression-List Right-Functor-Bracket
         Functor-Identifier "(" [ Term-Expression-List ] ")"
          Structure-Symbol "(#" Term-Expression-List "#)"
          "the" Structure-Symbol "of" Term-Expression
          Variable-Identifier
          "{" Term-Expression { Postqualification } ":" Sentence "}"
          "the" "set" "of" "all" Term-Expression { Postqualification }
         Numeral
          Term-Expression "qua" Type-Expression
          "the" Selector-Symbol "of" Term-Expression
          "the" Selector-Symbol
          "the" Type-Expression
          Private-Definition-Parameter
          "it" .
But I think it might be clearer if we view it using the equivalent grammar:
      Term-Expression = "(" Term-Expression ")"
          [ Arguments ] Functor-Symbol [ Arguments ]
         Left-Functor-Bracket Term-Expression-List Right-Functor-Bracket
         Functor-Identifier "(" [ Term-Expression-List ] ")"
          Aggregate-Term
          Forgetful-Functor-Term
          Variable-Identifier
          Fraenkel-Term
          Numeral
          Qualified-Term
          Selector-Functor
          Internal-Selector-Functor
          Choice-Term
         Private-Definition-Parameter
      Aggregate-Term = Structure-Symbol "(#" Term-Expression-List "#)" .
      Choice-Term = "the" Type-Expression.
      Forgetful-Functor-Term = "the" Structure-Symbol "of" Term-Expression.
      Fraenkel-Term = "{" Term-Expression {Postqualification} ":" Sentence "}"
        "the" "set" "of" "all" Term-Expression { Postqualification }.
      Internal-Selector-Functor = "the" Selector-Symbol.
      Selector-Functor = "the" Selector-Symbol "of" Term-Expression.
      Qualified-Term = Term-Expression "qua" Type-Expression.
```

```
755.
       Class structure for this syntax tree.
\langle Classes for terms (abstract syntax tree) 755\rangle \equiv
     { Terms }
  (Simple term (abstract syntax tree) 756);
  ⟨ Placeholder term (abstract syntax tree) 758⟩;
   Numeral term (abstract syntax tree) 760);
   Infix term (abstract syntax tree) 762;
   Terms with arguments (abstract syntax tree) 764);
   Circumfix term (abstract syntax tree) 766);
   Private functor term (abstract syntax tree) 768):
   One-argument term (abstract syntax tree) 770;
   Selector term (abstract syntax tree) 772;
   Internal selector term (abstract syntax tree) 774);
   Aggregate term (abstract syntax tree) 776);
   Forgetful functor (abstract syntax tree) 778);
   (Internal forgetful functors (abstract syntax tree) 780);
   Fraenkel terms (abstract syntax tree) 782);
   Exactly term (abstract syntax tree) 788;
   Qualified term (abstract syntax tree) 786);
   Choice term (abstract syntax tree) 790);
   \langle "It" term (abstract syntax tree) 792\rangle;
  (Incorrect term (abstract syntax tree) 794);
This code is used in section 737.
       Simple terms. Mizar describes variables as terms as a Simple Term.
\langle \text{Simple term (abstract syntax tree) } 756 \rangle \equiv
  Simple TermPtr = \uparrow Simple TermObj;
  Simple Term Obj = \mathbf{object} \ (Term Expression Obj)
     nIdent: integer; { identifier number }
     constructor Init(const aPos: Position; aIdentNr: integer);
  end
This code is used in section 755.
757. Constructors.
\langle \text{Implementing term AST 757} \rangle \equiv
constructor SimpleTermObj.Init(const aPos: Position; aIdentNr: integer);
  begin nTermPos \leftarrow aPos; nTermSort \leftarrow wsSimpleTerm; nIdent \leftarrow aIdentNr;
  end:
See also sections 759, 761, 763, 765, 767, 769, 771, 773, 775, 777, 779, 781, 783, 785, 787, 789, 791, 793, and 795.
This code is used in section 736.
758. Placeholder terms. These are the parameters "$1", "$2", etc., which appear in a private functor
"deffunc Foo(object) = ...".
\langle \text{Placeholder term (abstract syntax tree) } 758 \rangle \equiv
  PlaceholderTermPtr = \uparrow PlaceholderTermObj; \{ placeholder \}
  PlaceholderTermObj = \mathbf{object} \ (TermExpressionObj)
     nLocusNr: integer; \{ \$1, \dots \}
     constructor Init(const aPos: Position; aLocusNr: integer);
  end
This code is used in section 755.
```

759. Constructor.

```
\langle Implementing term AST 757\rangle += constructor PlaceholderTermObj.Init (const aPos: Position; aLocusNr: integer); begin nTermPos \leftarrow aPos; nTermSort \leftarrow wsPlaceholderTerm; nLocusNr \leftarrow aLocusNr; end;
```

760. Numeral terms. Mizar can handle 32-bit integers. If we wanted to extend this to, say, arbitrary precision arithmetic, then we would want to modify this class (and a few other places).

```
\langle \text{Numeral term (abstract syntax tree)} \ 760 \rangle \equiv Numeral TermPtr = \uparrow Numeral TermObj;
Numeral TermObj = \mathbf{object} \ (TermExpressionObj)
nValue: integer;
\mathbf{constructor} \ Init(\mathbf{const} \ aPos: \ Position; \ aValue: integer);
\mathbf{end}
```

This code is used in section 755.

761. Constructor.

```
\langle Implementing term AST 757\rangle +\equiv constructor NumeralTermObj.Init(const aPos: Position; aValue: integer); begin nTermPos \leftarrow aPos; nTermSort \leftarrow wsNumeralTerm; nValue \leftarrow aValue; end;
```

762. Infix terms. When we have infix binary operators, they are terms with arguments on both sides of it. For example x + 2 will have "+" be an infix term with arguments (x, 2).

We could permit multiple arguments on the left-hand side (and on the right-hand side), but they are comma-separated in Mizar. This could happen in finite group theory, for example, "p -signalizer_over H,G" has two arguments on the right but only one argument on the left.

```
\langle \operatorname{Infix} \operatorname{term} (\operatorname{abstract} \operatorname{syntax} \operatorname{tree}) \ 762 \rangle \equiv Infix Term Ptr = \uparrow Infix Term Obj;
Infix Term Obj = \operatorname{object} (Term Expression Obj)
nFunctor Symbol: integer;
nLeft Args, nRight Args: PList;
\operatorname{constructor} Init(\operatorname{const} \ aPos: \ Position; \ aFunctor Nr: integer; \ aLeft Args, \ aRight Args: \ PList);
\operatorname{destructor} \ Done; \ virtual;
end
```

This code is used in section 755.

763. Constructor.

```
⟨ Implementing term AST 757⟩ +≡
constructor InfixTermObj.Init(const aPos: Position;
aFunctorNr: integer;
aLeftArgs, aRightArgs: PList);
begin nTermPos \leftarrow aPos; nTermSort \leftarrow wsInfixTerm; nFunctorSymbol \leftarrow aFunctorNr;
nLeftArgs \leftarrow aLeftArgs; nRightArgs \leftarrow aRightArgs;
end;
destructor InfixTermObj.Done;
begin dispose(nLeftArgs, Done); dispose(nRightArgs, Done);
end;
```

end;

Terms with arguments. This class seems to be used only internally to the abstract_syntax.pas module. Recalling the UML class diagram (§753), we remember there are three sublcasses to this: private functor terms (which appear in Mizar when we use "deffunc F(...) = ..."), circumfix ("bracketed") terms, and aggregate terms (when we construct an instance of a structure). \langle Terms with arguments (abstract syntax tree) $764 \rangle \equiv$ $TermWithArgumentsPtr = \uparrow TermWithArgumentsObj;$ $TermWithArgumentsObj = \mathbf{object} \ (TermExpressionObj)$ **constructor** *Init*(**const** *aPos*: *Position*; *aKind*: *TermSort*; *aArgs*: *PList*); **destructor** Done: virtual: end This code is used in section 755. 765. Constructor. $\langle \text{Implementing term AST 757} \rangle + \equiv$ constructor TermWithArgumentsObj.Init(const aPos: Position; aKind: TermSort; aArgs: PList); **begin** $nTermPos \leftarrow aPos$; $nTermSort \leftarrow aKind$; $nArgs \leftarrow aArgs$; end: **destructor** Term With Arguments Obj. Done; **begin** dispose(nArgs, Done); end: 766. Circumfix terms. We can introduce different types of brackets in Mizar. For example, for groups, we have the commutator of group elements [.x,y.]. These "bracketed terms" are referred to as circumfix terms. $\langle \text{Circumfix term (abstract syntax tree) 766} \rangle \equiv$ $CircumfixTermPtr = \uparrow CircumfixTermObj;$ $CircumfixTermObj = \mathbf{object} \ (TermWithArgumentsObj)$ nLeftBracketSymbol, nRightBracketSymbol: integer;**constructor** Init(**const** aPos: Position; aLeftBracketNr, aRightBracketNr: integer; aArgs: PList); **destructor** Done; virtual; end This code is used in section 755. 767. Constructor. $\langle \text{Implementing term AST 757} \rangle + \equiv$ **constructor** CircumfixTermObj.Init(**const** aPos: Position; aLeftBracketNr, aRightBracketNr: integer; aArgs: PList);**begin** inherited $Init(aPos, wsCircumfixTerm, aArgs); nLeftBracketSymbol <math>\leftarrow aLeftBracketNr;$ $nRightBracketSymbol \leftarrow aRightBracketNr;$ end: **destructor** CircumfixTermObj.Done; **begin** dispose(nArgs, Done);

```
768. Private functor terms. We introduce private functor terms in Mizar when we have "defpred
F(\dots) = \dots
\langle Private functor term (abstract syntax tree) 768 \rangle \equiv
  PrivateFunctorTermPtr = \uparrow PrivateFunctorTermObj;
  PrivateFunctorTermObj = \mathbf{object} \ (TermWithArgumentsObj)
    nFunctorIdent: integer;
    constructor Init (const aPos: Position; aFunctorIdNr: integer; aArgs: PList);
    destructor Done; virtual;
  end
This code is used in section 755.
769. Constructor.
\langle \text{Implementing term AST 757} \rangle + \equiv
constructor PrivateFunctorTermObj.Init(const aPos: Position; aFunctorIdNr: integer; aArqs: PList);
  begin inherited\ Init(aPos, wsPrivateFunctorTerm, aArgs);\ nFunctorIdent \leftarrow aFunctorIdNr;
destructor PrivateFunctorTermObj.Done;
  begin dispose(nArgs, Done);
  end;
       One-argument terms. Recalling the UML class diagram for terms (§753), we remember the class
for OneArgument terms are either selector terms ("the \langle field\rangle of \langle aggregate\rangle") or forgetful functors ("the
\langle structure \rangle of \langle aggregate \rangle").
\langle \text{One-argument term (abstract syntax tree) } 770 \rangle \equiv
  OneArgumentTermPtr = \uparrow OneArgumentTermObj;
  OneArgumentTermObj = object (TermExpressionObj)
    nArg: TermPtr;
    constructor Init (const aPos: Position; aKind: TermSort; aArg: TermPtr);
    destructor Done; virtual;
  end
This code is used in section 755.
771. Constructor.
\langle \text{Implementing term AST 757} \rangle + \equiv
constructor One Argument TermObj. Init(const aPos: Position; aKind: TermSort; aArg: TermPtr);
  begin nTermPos \leftarrow aPos; nTermSort \leftarrow aKind; nArg \leftarrow aArg;
  end;
destructor OneArgumentTermObj.Done;
  begin dispose(nArg, Done);
  end;
```

258

772. Selector terms. When we have an aggregate term (i.e., an instance of a structure), we want to refer to fields of the structure. This is done with selector terms. [The selector number refers to the position in the underlying tuple of the structure instance.]

```
\langle Selector term (abstract syntax tree) 772\rangle \equiv
  SelectorTermPtr = \uparrow SelectorTermObj;
  SelectorTermObj = \mathbf{object} \ (OneArgumentTermObj)
     nSelectorSymbol: integer;
     constructor Init(const aPos: Position; aSelectorNr: integer; aArq: TermPtr);
     destructor Done; virtual;
  end
This code is used in section 755.
773. Constructor.
\langle \text{Implementing term AST 757} \rangle + \equiv
constructor SelectorTermObj.Init(const aPos: Position; aSelectorNr: integer; aArq: TermPtr);
  begin inherited Init(Apos, wsSelectorTerm, aArg); nSelectorSymbol <math>\leftarrow aSelectorNr;
  end:
destructor SelectorTermObj.Done;
  begin dispose(nArg, Done);
  end:
774. Internal selector terms. An "internal selector" term refers to the case where we have in Mizar
"the \langle selector \rangle" treated as a term.
\langle \text{Internal selector term (abstract syntax tree) } 774 \rangle \equiv
  InternalSelectorTermPtr = \uparrow InternalSelectorTermObj;
  InternalSelectorTermObj = \mathbf{object} \ (TermExpressionObj)
     nSelectorSymbol: integer;
     constructor Init(const aPos: Position; aSelectorNr: integer);
  end
This code is used in section 755.
775.
       Constructor.
\langle \text{Implementing term AST 757} \rangle + \equiv
constructor InternalSelectorTermObj.Init(const aPos: Position; aSelectorNr: integer);
  begin nTermPos \leftarrow aPos; nTermSort \leftarrow wsInternalSelectorTerm; nSelectorSymbol \leftarrow aSelectorNr;
  end:
       Aggregate terms. When we construct a new instance of a structure, well, that's a term. Such
terms are called "aggregate terms" in Mizar.
\langle \text{Aggregate term (abstract syntax tree) } 776 \rangle \equiv
  AggregateTermPtr = \uparrow AggregateTermObj;
  AggregateTermObj = \mathbf{object} \ (TermWithArgumentsObj)
     nStructSymbol: integer;
     constructor Init(const aPos: Position; aStructSymbol: integer; aArgs: PList);
     destructor Done; virtual;
  end
This code is used in section 755.
```

```
777.
       Constructor.
\langle \text{Implementing term AST 757} \rangle + \equiv
constructor Aggregate TermObj. Init(const aPos: Position; aStructSymbol: integer; aArgs: PList);
  begin inherited Init(aPos, wsAqqreqateTerm, aArqs); nStructSymbol <math>\leftarrow aStructSymbol;
  end:
destructor Aggregate TermObj.Done;
  begin dispose(nArgs, Done);
  end:
778. Forgetful functors. When we have structure inheritance in Mizar, say structure B extends struc-
ture A, and we have b being an instance of B, then we can obtain "the A-object underlying b" by writing
"the A of b". This is an example of what Mizar calls a "forgetful functor" (which is quite the pun).
\langle Forgetful functor (abstract syntax tree) 778\rangle \equiv
  ForgetfulFunctorTermPtr = \uparrow ForgetfulFunctorTermObj;
  ForgetfulFunctorTermObj = \mathbf{object} \ (OneArgumentTermObj)
    nStructSymbol: integer;
    constructor Init(const aPos: Position; aStructSymbol: integer; aArq: TermPtr);
    destructor Done; virtual;
  end
This code is used in section 755.
      Constructor.
\langle \text{Implementing term AST 757} \rangle + \equiv
constructor ForgetfulFunctorTermObj.Init(const aPos: Position; aStructSymbol: integer;
                                               aArg: TermPtr);
  begin inherited Init(aPos, wsForgetfulFunctorTerm, aArg); nStructSymbol <math>\leftarrow aStructSymbol;
  end;
destructor ForgetfulFunctorTermObj.Done;
  begin dispose(nArg, Done);
  end;
780. Internal forgetful functors. When we omit the "structure instance" b in a forgetful functor term
— e.g., when we have "the A" — then we have an "internal forgetful functor" (named analogous to internal
selectors).
\langle \text{Internal forgetful functors (abstract syntax tree) } 780 \rangle \equiv
  InternalForgetfulFunctorTermPtr = \uparrow InternalForgetfulFunctorTermObj;
  InternalForgetfulFunctorTermObj = \mathbf{object} \ (TermExpressionObj)
    nStructSymbol: integer;
    constructor Init (const aPos: Position; aStructSymbol: integer);
  end
This code is used in section 755.
781. Constructor.
\langle \text{Implementing term AST 757} \rangle + \equiv
constructor InternalForgetfulFunctorTermObj.Init(const aPos: Position; aStructSymbol: integer);
  begin nTermPos \leftarrow aPos; nTermSort \leftarrow wsInternalForgetfulFunctorTerm;
  nStructSymbol \leftarrow aStructSymbol;
  end;
```

Mizar Parser

260

Simple Fraenkel terms. Fraenkel terms are set-builder notation in Mizar. But "simple" Fraenkel terms occurs when we have "the set of all $\langle termexpr \rangle$ ". \langle Fraenkel terms (abstract syntax tree) $782 \rangle \equiv$ $SimpleFraenkelTermPtr = \uparrow SimpleFraenkelTermObj;$ $SimpleFraenkelTermObj = \mathbf{object} \ (TermExpressionObj)$ $nPostqualification: PList; { of segments }$ nSample: TermPtr;**constructor** Init(**const** aPos: Position; aPostqual: PList; aSample: TermPtr); destructor Done; virtual; end:See also section 784. This code is used in section 755. 783. Constructor. $\langle \text{Implementing term AST 757} \rangle + \equiv$ **constructor** SimpleFraenkelTermObj.Init(**const** aPos: Position; aPostqual: PList; aSample: TermPtr); **begin** $nTermPos \leftarrow aPos$; $nTermSort \leftarrow wsSimpleFraenkelTerm$; $nPostqualification \leftarrow aPostqual$; $nSample \leftarrow aSample;$ end; **destructor** SimpleFraenkelTermObj.Done; **begin** dispose(nSample, Done);end; 784. Fraenkel terms. Fraenkel terms are sets given by set-builder notation, usually they look like $\{f(\vec{t}) \text{ where } \vec{t} \text{ being } \vec{T} : P[\vec{t}]\}$ This is technically a higher-order object (look, it takes a functor f and a predicate P). \langle Fraenkel terms (abstract syntax tree) 782 $\rangle + \equiv$ $FraenkelTermPtr = \uparrow FraenkelTermObj;$ $FraenkelTermObj = \mathbf{object} \ (SimpleFraenkelTermObj)$ *nFormula: FormulaPtr*; constructor Init(const aPos: Position; aPostqual: PList; aSample: TermPtr; aFormula: FormulaPtr);**destructor** Done; virtual; end

785. Constructor.

```
\langle \text{Implementing term AST 757} \rangle + \equiv
constructor FraenkelTermObj.Init(const aPos: Position;
                                        aPostqual: PList;
                                        aSample: TermPtr;
                                        aFormula: FormulaPtr);
  begin nTermPos \leftarrow aPos; nTermSort \leftarrow wsFraenkelTerm; nPostqualification \leftarrow aPostqual;
  nSample \leftarrow aSample; nFormula \leftarrow aFormula;
  end;
destructor FraenkelTermObj.Done;
  begin dispose(nSample, Done); dispose(nPostqualification, Done); dispose(nFormula, Done);
  end;
```

begin dispose(nSubject, Done);

end;

```
Qualified terms. We may wish to explicitly type cast a term (e.g., "term qua newType"), which
is what Mizar calls a "qualified term".
\langle \text{Qualified term (abstract syntax tree) } 786 \rangle \equiv
  Qualified TermPtr = \uparrow Qualified TermObj;
  QualifiedTermObj = \mathbf{object} (ExactlyTermObj)
     nQualification: TypePtr;
     constructor Init(const aPos: Position; aSubject: TermPtr; aType: TypePtr);
     destructor Done; virtual;
  end
This code is used in section 755.
787.
       Constructor.
\langle \text{Implementing term AST 757} \rangle + \equiv
constructor Qualified TermObj.Init(const aPos: Position; aSubject: TermPtr; aType: TypePtr);
  begin nTermPos \leftarrow aPos; nTermSort \leftarrow wsQualificationTerm; nSubject \leftarrow aSubject;
  nQualification \leftarrow aType;
  end:
destructor Qualified Term Obj. Done;
  begin dispose(nSubject, Done); dispose(nQualification, Done);
  end;
788. Exactly terms. This is the base class for qualified terms. It does not appear to be used anywhere
outside the abstract syntax module.
\langle \text{Exactly term (abstract syntax tree) 788} \rangle \equiv
  Exactly TermPtr = \uparrow Exactly TermObj;
  Exactly TermObj = \mathbf{object} \ (TermExpressionObj)
     nSubject: TermPtr;
     constructor Init(const aPos: Position; aSubject: TermPtr);
     destructor Done; virtual;
  end
This code is used in section 755.
789. Constructor.
\langle \text{Implementing term AST 757} \rangle + \equiv
constructor Exactly TermObj. Init(const aPos: Position; aSubject: TermPtr);
  begin nTermPos \leftarrow aPos; nTermSort \leftarrow wsExactlyTerm; nSubject \leftarrow aSubject;
  end;
destructor ExactlyTermObj.Done;
```

constructor $IncorrectTermObj.Init(\mathbf{const}\ aPos:\ Position);$ **begin** $nTermPos \leftarrow aPos;\ nTermSort \leftarrow wsErrorTerm;$

end;

Choice terms. This refers to "the $\langle type \rangle$ " terms. It is a "global choice term" of sorts, except it 790. "operates" on soft types instead of arbitrary predicates. \langle Choice term (abstract syntax tree) $790 \rangle \equiv$ $Choice TermPtr = \uparrow Choice TermObj;$ $ChoiceTermObj = \mathbf{object} \ (TermExpressionObj)$ nChoiceType: TypePtr;**constructor** *Init* (**const** *aPos*: *Position*; *aType*: *TypePtr*); **destructor** Done; virtual; end This code is used in section 755. 791. Constructor. $\langle \text{Implementing term AST 757} \rangle + \equiv$ **constructor** Choice Term Obj. Init (**const** a Pos: Position; a Type: TypePtr); **begin** $nTermPos \leftarrow aPos; nTermSort \leftarrow wsGlobalChoiceTerm; nChoiceType \leftarrow aType;$ end; **destructor** Choice Term Obj. Done; **begin** dispose(nChoiceType, Done);end; 792. It terms. When we define a new mode [type] or functors [terms], Mizar introduces an anaphoric keyword "it" referring to an example of the mode (resp., to the term being defined). Here I borrow the scary phrase "anaphoric" from Lisp macros, so blame Paul Graham for this pretentiousness. \langle "It" term (abstract syntax tree) $792 \rangle \equiv$ $ItTermPtr = \uparrow ItTermObj;$ $ItTermObj = \mathbf{object} \ (TermExpressionObj)$ **constructor** *Init* (**const** *aPos*: *Position*); end This code is used in section 755. 793. Constructor. $\langle \text{Implementing term AST 757} \rangle + \equiv$ constructor ItTermObj.Init(const aPos: Position); **begin** $nTermPos \leftarrow aPos; nTermSort \leftarrow wsItTerm;$ end; 794. Incorrect terms. Generically, when we run into an error of some kind, we represent the term with an Incorrect term instance. This will allow Mizar to continue working when the user goofed. $\langle \text{Incorrect term (abstract syntax tree) } 794 \rangle \equiv$ $IncorrectTermPtr = \uparrow IncorrectTermObj;$ $IncorrectTermObj = \mathbf{object} \ (TermExpressionObj)$ **constructor** *Init* (**const** *aPos*: *Position*); end This code is used in section 755. **795**. Constructor. $\langle \text{Implementing term AST 757} \rangle + \equiv$

This code is used in section 737.

Section 20.2. TYPES (ABSTRACT SYNTAX TREE)

```
The grammar for Mizar types looks like:
       Type-Expression = "(" Radix-Type ")"
            Adjective-Cluster Type-Expression
           Radix-Type .
       Structure-Type-Expression =
            "(" Structure-Symbol ["over" Term-Expression-List] ")"
           Adjective-Cluster Structure-Symbol [ "over" Term-Expression-List ].
       Radix-Type = Mode-Symbol [ "of" Term-Expression-List ]
          | Structure-Symbol [ "over" Term-Expression-List ] .
       Type-Expression-List = Type-Expression { "," Type-Expression } .
So there are several main sources of modes [types]: structures, primitive types (like "set" and "object"),
and affixing adjectives to types.
  For readers who are unfamiliar with types in Mizar, they are "soft types". What does this mean?
Well, we refer the reader to Free Wiedijk's "Mizar's Soft Type System" (in K. Schneider and J. Brandt,
eds., Theorem Proving in Higher Order Logics. TPHOLs 2007, Springer, doi:10.1007/978-3-540-74591-
4_28). Essentially, a type ascription in Mizar of the form "for x being Foo st P[x] holds Q[x]", this is
equivalent to Foo being a unary predicate and the formula in first-order logic is "\forall x. Foo [x] \land Q[x] \implies P[x]".
      We have an abstract base class for types.
\langle \text{ Abstract base class for types 797} \rangle \equiv
  TypeSort = (wsErrorType, wsStandardType, wsStructureType, wsClusteredType, wsReservedDscrType);
       { Initial structures }
  TypePtr = \uparrow TypeExpressionObj;
  TypeExpressionObj = \mathbf{object} (MObject)
    nTypeSort: TypeSort;
    nTypePos: Position;
    end
This code is used in section 737.
798. Radix type. A "radix type" refers to any type of the form "\langle RadixType \rangle of T_1, \ldots, T_n". This
usually appears when defining a new expandable mode, where we have:
     "mode \langle Expandable\ Mode \rangle is \langle Adjective_1 \rangle \ldots \langle Adjective_n \rangle \langle Radix\ Type \rangle"
This appears to be used only in definitions.
\langle Classes for type (abstract syntax tree) 798\rangle \equiv
    { Types }
  RadixTypePtr = \uparrow RadixTypeObj;
  RadixTypeObj = \mathbf{object} (TypeExpressionObj)
    nArgs: PList; \{ of \}
    constructor Init(const aPos: Position; aKind: TypeSort; aArgs: PList);
    destructor Done; virtual;
  end:
See also sections 800, 802, 804, and 806.
```

end;

```
264
                                                                                           Mizar Parser
799.
       Constructor.
\langle \text{Implementing type AST 799} \rangle \equiv
constructor RadixTypeObj.Init(const aPos: Position; aKind: TypeSort; aArgs: PList);
  begin nTypePos \leftarrow aPos; nTypeSort \leftarrow aKind; nArgs \leftarrow aArgs;
  end:
destructor RadixTypeObj.Done;
  begin dispose(nArgs, Done);
See also sections 801, 803, 805, and 807.
This code is used in section 736.
       Standard type. When we want to refer to an expandable mode in a Mizar formula, then it is
represented by a "standard type". This contrasts it with "clustered types" (i.e., a type stacked with
adjectives) and "structure types".
\langle Classes for type (abstract syntax tree) 798\rangle + \equiv
  StandardTypePtr = \uparrow StandardTypeObj;
  StandardTypeObj = \mathbf{object} (RadixTypeObj)
    nModeSymbol: integer;
    constructor Init(const aPos: Position; aModeSymbol: integer; aArgs: PList);
    destructor Done; virtual;
  end;
801. Constructor.
\langle \text{Implementing type AST 799} \rangle + \equiv
constructor StandardTypeObj.Init(const aPos: Position; aModeSymbol: integer; aArgs: PList);
  begin inherited Init(aPos, wsStandardType, aArgs); nModeSymbol <math>\leftarrow aModeSymbol;
  end;
destructor Standard Type Obj. Done;
  begin inherited Done;
  end;
      Structure type. When we define a new structure, we are really introducing a new type.
aArgs tracks its parent structures and parameter types. The structure type extends the RadixType class
because RadixType instances can be "stacked with adjectives".
\langle Classes for type (abstract syntax tree) 798\rangle + \equiv
  StructTypePtr = \uparrow StructTypeObj;
  StructTypeObj = \mathbf{object} (RadixTypeObj)
    nStructSymbol: integer;
    constructor Init(const aPos: Position; aStructSymbol: integer; aArgs: PList);
    destructor Done; virtual;
  end:
803. Constructor.
\langle Implementing type AST 799\rangle +\equiv
constructor StructTypeObj.Init(const aPos: Position; aStructSymbol: integer; aArqs: PList);
  begin inherited Init(aPos, wsStructureType, aArgs); nStructSymbol <math>\leftarrow aStructSymbol;
  end;
destructor StructTypeObj.Done;
  begin inherited Done;
```

804. Clustered type. The clustered type describes the situation where we accumulate aCluster of adjectives atop aType.

```
\langle Classes for type (abstract syntax tree) 798\rangle + \equiv
  ClusteredTypePtr = \uparrow ClusteredTypeObj;
  ClusteredTypeObj = \mathbf{object} \ (TypeExpressionObj)
     nAdjectiveCluster: PList;
     nType: TypePtr;
     constructor Init (const aPos: Position; aCluster: PList; aType: TypePtr);
     destructor Done; virtual;
  end;
805. Constructor.
\langle \text{Implementing type AST 799} \rangle + \equiv
constructor Clustered Type Obj. Init (const a Pos: Position; a Cluster: PList; a Type: Type Ptr);
  begin nTypePos \leftarrow aPos; nTypeSort \leftarrow wsClusteredType; nAdjectiveCluster \leftarrow aCluster;
  nType \leftarrow aType;
  end;
destructor Clustered Type Obj. Done;
  begin dispose(nAdjectiveCluster, Done); dispose(nType, Done);
  end;
806. Incorrect type. We want Mizar to be resilient against typing errors, so we have an IncorrectType
node for the syntax tree. The alternative would be to crash upon error.
\langle Classes for type (abstract syntax tree) 798\rangle + \equiv
  IncorrectTypePtr = \uparrow IncorrectTypeObj;
  IncorrectTypeObj = \mathbf{object} (TypeExpressionObj)
     constructor Init(const aPos: Position);
  end
807. Constructor.
\langle \text{Implementing type AST 799} \rangle + \equiv
constructor IncorrectTypeObj.Init(const aPos: Position);
  begin nTypePos \leftarrow aPos; nTypeSort \leftarrow wsErrorType;
  end;
```

Section 20.3. FORMULAS (ABSTRACT SYNTAX TREE)

```
We have an abstract base class for formulas.
\langle Abstract base class for formulas 808 \rangle \equiv
    FormulaSort = (wsErrorFormula, wsThesis, wsContradiction, wsRightSideOfPredicativeFormula, wsContradiction, wsRightSideOfPredicativeFormula, wsContradiction, w
             wsPredicativeFormula, wsMultiPredicativeFormula, wsPrivatePredicateFormula,
             wsAttributiveFormula, wsQualifyingFormula, wsUniversalFormula, wsExistentialFormula,
             wsNegatedFormula, wsConjunctiveFormula, wsDisjunctiveFormula, wsConditionalFormula,
             wsBiconditional Formula, wsFlexaryConjunctiveFormula, wsFlexaryDisjunctiveFormula);
    FormulaPtr = \uparrow FormulaExpressionObj;
    FormulaExpressionObj = \mathbf{object} (MObject)
         nFormulaSort: FormulaSort;
         nFormulaPos: Position;
         end
This code is used in section 737.
809. The syntax for Mizar formulas looks like:
    Formula-Expression = "(" Formula-Expression ")"
            Atomic-Formula-Expression
            Quantified-Formula-Expression
            Formula-Expression "&" Formula-Expression
            Formula-Expression "&" "..." "&" Formula-Expression
            Formula-Expression "or" Formula-Expression
            Formula-Expression "or" "..." "or" Formula-Expression
            Formula-Expression "implies" Formula-Expression
            Formula-Expression "iff" Formula-Expression
            "not" Formula-Expression
            "contradiction"
            "thesis" .
    Atomic-Formula-Expression =
            [Term-Expression-List] [("does" | "do") "not"] Predicate-Symbol [Term-Expression-List]
            {[("does" | "do") "not"] Predicate-Symbol Term-Expression-List}
            Predicate-Identifier "[" [ Term-Expression-List ] "]"
            Term-Expression "is" Adjective { Adjective }
            Term-Expression "is" Type-Expression .
    Quantified-Formula-Expression =
                  "for" Qualified-Variables
                  [ "st" Formula-Expression ]
                  ( "holds" Formula-Expression | Quantified-Formula-Expression )
         "ex" Qualified-Variables "st" Formula-Expression .
```

```
810. Right-side of predicative formula.
\langle Classes for formula (abstract syntax tree) 810 \rangle \equiv
     { Formulas }
  RightSideOfPredicativeFormulaPtr = \uparrow RightSideOfPredicativeFormulaObj;
  RightSideOfPredicativeFormulaObj = \mathbf{object} (FormulaExpressionObj)
     nPredNr: integer;
     nRightArgs: PList;
     constructor Init(const aPos: Position; aPredNr: integer; aRightArgs: PList);
     destructor Done; virtual;
  end
See also sections 812, 814, 816, 818, 820, 822, 824, 826, 828, 830, 832, 834, 836, 838, 840, 842, 844, 846, and 848.
This code is used in section 737.
811.
       Constructor.
\langle \text{Implementing formula AST 811} \rangle \equiv
constructor RightSideOfPredicativeFormulaObj.Init(const aPos: Position;
                                                           aPredNr: integer;
                                                           aRightArgs: PList);
  begin nFormulaPos \leftarrow aPos; nFormulaSort \leftarrow wsRightSideOfPredicativeFormula;
  nPredNr \leftarrow aPredNr; \ nRightArgs \leftarrow aRightArgs;
  end;
{\bf destructor}\ \textit{RightSideOfPredicativeFormulaObj.Done};
  begin dispose(nRightArgs, Done);
  end:
See also sections 813, 815, 817, 819, 821, 823, 825, 827, 829, 831, 833, 835, 837, 839, 841, 843, 845, 847, and 849.
This code is used in section 736.
812. Predicative formula. A "predicative" formula refers to a formula involving predicates. A predicate
will have a list of terms \vec{t} it expects as arguments, as well as two numbers \ell, r such that t_1, \ldots, t_\ell are the
arguments to its left, and t_{\ell+1}, \ldots, t_{\ell+r} are on the right. When \ell=0, all arguments are on the right; and
when r = 0, all arguments are on the left.
\langle Classes for formula (abstract syntax tree) 810 \rangle + \equiv
  PredicativeFormulaPtr = \uparrow PredicativeFormulaObj;
  PredicativeFormulaObj = \mathbf{object} \ (RightSideOfPredicativeFormulaObj)
     nLeftArgs: PList;
     constructor Init(const aPos: Position; aPredNr: integer; aLeftArgs, aRightArgs: PList);
     destructor Done; virtual;
  end
813.
       Constructor.
\langle \text{Implementing formula AST 811} \rangle + \equiv
constructor PredicativeFormulaObj.Init(const aPos: Position;
```

aPredNr: integer;

begin $nFormulaPos \leftarrow aPos; nFormulaSort \leftarrow wsPredicativeFormula; nPredNr \leftarrow aPredNr;$

 $nLeftArgs \leftarrow aLeftArgs; nRightArgs \leftarrow aRightArgs;$

begin dispose(nLeftArgs, Done); dispose(nRightArgs, Done);

destructor PredicativeFormulaObj.Done:

end:

end;

aLeftArgs, aRightArgs: PList);

814. Multi-predicative formula. The Working Mathematician writes things like " $1 \le i \le ||T||$ " and Mizar wants to support this. Multi-predicative formulas are of this form "1 <= i <= len T". This occurs in VECTSP13, for example.

```
\langle Classes for formula (abstract syntax tree) 810 \rangle + \equiv
  MultiPredicativeFormulaPtr = \uparrow MultiPredicativeFormulaObj;
  MultiPredicativeFormulaObj = \mathbf{object} (FormulaExpressionObj)
     nScraps: PList;
     constructor Init(const aPos: Position; aScraps: PList);
     destructor Done; virtual;
  end
815.
       Constructor.
\langle \text{Implementing formula AST 811} \rangle + \equiv
constructor MultiPredicativeFormulaObj.Init(const aPos: Position; aScraps: PList);
  begin nFormulaPos \leftarrow aPos; nFormulaSort \leftarrow wsMultiPredicativeFormula; nScraps <math>\leftarrow aScraps;
  end;
destructor MultiPredicativeFormulaObj.Done;
  begin dispose(nScraps, Done);
  end;
       Attributive formula. As part of Mizar's soft type system, we can use attributes (adjectives) to
form a formula like "\langle term \rangle is \langle adjective \rangle". We can stack multiple adjectives in an attributive formula.
\langle Classes for formula (abstract syntax tree) 810 \rangle +\equiv
  AttributiveFormulaPtr = \uparrow AttributiveFormulaObj;
  AttributiveFormulaObj = \mathbf{object} (FormulaExpressionObj)
     nSubject: TermPtr;
     nAdjectives: PList;
     constructor Init(const aPos: Position; aSubject: TermPtr; aAdjectives: PList);
     destructor Done; virtual;
  end
817. Constructor.
\langle \text{Implementing formula AST 811} \rangle + \equiv
constructor AttributiveFormulaObj.Init(const aPos: Position; aSubject: TermPtr; aAdjectives: PList);
  begin nFormulaPos \leftarrow aPos; nFormulaSort \leftarrow wsAttributiveFormula; nSubject \leftarrow aSubject;
  nAdjectives \leftarrow aAdjectives;
  end;
destructor AttributiveFormulaObj.Done;
  begin dispose(nSubject, Done); dispose(nAdjectives, Done);
  end:
```

end;

"P" as a private predicate. It is represented in the abstract syntax tree as a private predicative formula object. \langle Classes for formula (abstract syntax tree) 810 $\rangle + \equiv$ $Private Predicative Formula Ptr = \uparrow Private Predicative Formula Obj;$ $PrivatePredicativeFormulaObj = \mathbf{object} (FormulaExpressionObj)$ nPredIdNr: integer;nArgs: PList;**constructor** Init(**const** aPos: Position; aPredIdNr: integer; aArgs: PList); **destructor** Done: virtual: end 819. Constructor. $\langle \text{Implementing formula AST 811} \rangle + \equiv$ **constructor** PrivatePredicativeFormulaObj.Init(**const** aPos: Position; aPredIdNr: integer;aArgs: PList);**begin** $nFormulaPos \leftarrow aPos$; $nFormulaSort \leftarrow wsPrivatePredicateFormula$; $nPredIdNr \leftarrow aPredIdNr$; $nArgs \leftarrow aArgs;$ end; **destructor** PrivatePredicativeFormulaObj.Done; **begin** dispose(nArgs, Done);end: Qualifying formula. Using Mizar's soft type system, we may have formulas of the form "\langle term\rangle is $\langle type \rangle$ ". These are referred to as "qualifying formulas", at least when discussing the abstract syntax tree. \langle Classes for formula (abstract syntax tree) 810 \rangle + \equiv $QualifyingFormulaPtr = \uparrow QualifyingFormulaObj;$ $QualifyingFormulaObj = \mathbf{object} (FormulaExpressionObj)$ nSubject: TermPtr;nType: TypePtr;**constructor** Init(**const** aPos: Position; aSubject: TermPtr; aType: TypePtr); y destructor Done; virtual; end Constructor. $\langle \text{Implementing formula AST 811} \rangle + \equiv$ constructor QualifyingFormulaObj.Init(const aPos: Position; aSubject: TermPtr; aType: TypePtr); **begin** $nFormulaPos \leftarrow aPos; nFormulaSort \leftarrow wsQualifyingFormula; <math>nSubject \leftarrow aSubject;$ $nType \leftarrow aType$; end; **destructor** QualifyingFormulaObj.Done; **begin** dispose(nSubject, Done); dispose(nType, Done);

Private predicative formula. When we have "defpred P[...] means ..." in Mizar, we refer to

end

Negative formula. Now we can proceed with the familiar formulas in first-order logic. Negative formulas are of the form $\neg \varphi$ for some formula φ . \langle Classes for formula (abstract syntax tree) 810 \rangle + \equiv $NegativeFormulaPtr = \uparrow NegativeFormulaObj;$ $NegativeFormulaObj = \mathbf{object} (FormulaExpressionObj)$ nArg: FormulaPtr;**constructor** *Init* (**const** *aPos*: *Position*; *aArg*: *FormulaPtr*); **destructor** Done; virtual; end 823. Constructor. $\langle \text{Implementing formula AST 811} \rangle + \equiv$ **constructor** NegativeFormulaObj.Init(**const** aPos: Position; aArq: FormulaPtr); **begin** $nFormulaPos \leftarrow aPos; nFormulaSort \leftarrow wsNegatedFormula; <math>nArg \leftarrow aArg;$ end; **destructor** NegativeFormulaObj.Done; **begin** dispose(nArq, Done); end; 824. Binary arguments formula. We have a class describing formulas involving binary logical connectives. We will extend it to describe conjunctive formulas, disjunctive formulas, conditionals, biconditionals, etc. \langle Classes for formula (abstract syntax tree) 810 \rangle + \equiv $BinaryFormulaPtr = \uparrow BinaryArgumentsFormula;$ $BinaryArgumentsFormula = \mathbf{object} (FormulaExpressionObj)$ nLeftArg, nRightArg: FormulaPtr; **constructor** *Init* (**const** *aPos*: *Position*; *aLeftArg*, *aRightArg*: *FormulaPtr*); **destructor** Done; virtual; end 825. Constructor. $\langle \text{Implementing formula AST 811} \rangle + \equiv$ **constructor** BinaryArgumentsFormula.Init(**const** aPos: Position; aLeftArq, aRightArq: FormulaPtr); $\textbf{begin} \ nFormulaPos \leftarrow aPos; \ nLeftArg \leftarrow aLeftArg; \ nRightArg \leftarrow aRightArg;$ end; **destructor** BinaryArgumentsFormula.Done; **begin** dispose(nLeftArg, Done); dispose(nRightArg, Done);end; 826. Conjunctive formula. A conjunctive formula looks like $\varphi \wedge \psi$ where φ and ψ are logical formulas. \langle Classes for formula (abstract syntax tree) 810 \rangle + \equiv $ConjunctiveFormulaPtr = \uparrow ConjunctiveFormulaObj;$ $ConjunctiveFormulaObj = \mathbf{object} \ (BinaryArgumentsFormula)$

constructor Init(**const** aPos: Position; aLeftArq, aRightArq: FormulaPtr);

827. Constructor.

```
⟨Implementing formula AST 811⟩ +≡
constructor ConjunctiveFormulaObj.Init(const aPos: Position; aLeftArg, aRightArg: FormulaPtr);
begin inheritedInit(aPos, aLeftArg, aRightArg); nFormulaSort ← wsConjunctiveFormula;
end;
828. Disjunctive formula. Disjunctive formulas look like φ ∨ ψ where φ and ψ are formulas.
⟨Classes for formula (abstract syntax tree) 810⟩ +≡
DisjunctiveFormulaPtr = ↑DisjunctiveFormulaObj;
DisjunctiveFormulaObj = object (BinaryArgumentsFormula)
```

829. Constructor.

end

```
\langle Implementing formula AST 811\rangle +\equiv constructor DisjunctiveFormulaObj.Init(const aPos: Position; aLeftArg, aRightArg: FormulaPtr); begin inheritedInit(aPos, aLeftArg, aRightArg); nFormulaSort \leftarrow wsDisjunctiveFormula; end;
```

constructor Init(**const** aPos: Position; aLeftArg, aRightArg: FormulaPtr);

830. Conditional formula. Conditional formulas look like $\varphi \implies \psi$ where φ and ψ are formulas.

```
\langle Classes for formula (abstract syntax tree) 810 \rangle +\equiv ConditionalFormulaPtr = \uparrow ConditionalFormulaObj; ConditionalFormulaObj = object (BinaryArgumentsFormula) constructor Init(const aPos: Position; aLeftArg, aRightArg: FormulaPtr); end
```

831. Constructor.

```
\langle Implementing formula AST 811\rangle +\equiv constructor ConditionalFormulaObj.Init(const aPos: Position; aLeftArg, aRightArg: FormulaPtr); begin inheritedInit(aPos, aLeftArg, aRightArg); nFormulaSort \leftarrow wsConditionalFormula; end:
```

832. Biconditional formula. Biconditional formulas look like $\varphi \iff \psi$ where φ and ψ are formulas.

```
\langle \text{Classes for formula (abstract syntax tree) } 810 \rangle + \equiv BiconditionalFormulaPtr = \uparrow BiconditionalFormulaObj; 
BiconditionalFormulaObj = object (BinaryArgumentsFormula) 
constructor Init(const aPos: Position; aLeftArg, aRightArg: FormulaPtr); end
```

833. Constructor.

```
\langle \text{Implementing formula AST 811} \rangle + \equiv  constructor BiconditionalFormulaObj.Init(\mathbf{const}\ aPos:\ Position;\ aLeftArg,\ aRightArg:\ FormulaPtr); begin inherited\ Init(aPos,\ aLeftArg,\ aRightArg);\ nFormulaSort \leftarrow wsBiconditionalFormula; end;
```

Mizar Parser

834. Flexary Conjunctive formula. Flexary conjunctive formulas are unique to Mizar, though the Working Mathematician would recognize them as "just a bunch of conjunctions". These look like $\varphi[1] \wedge \cdots \wedge \varphi[n]$ where $\varphi[i]$ is a formula parametrized by a natural number i.

836. Flexary Disjunctive formula. Flexary disjunctive formulas are unique to Mizar, though the Working Mathematician would recognize them as "just a bunch of disjunctions". These look like $\varphi[1] \vee \cdots \vee \varphi[n]$ where $\varphi[i]$ is a formula parametrized by a natural number i.

```
⟨ Classes for formula (abstract syntax tree) 810⟩ +≡
FlexaryDisjunctiveFormulaPtr = ↑FlexaryDisjunctiveFormulaObj;
FlexaryDisjunctiveFormulaObj = object (BinaryArgumentsFormula)
constructor Init(const aPos: Position; aLeftArg, aRightArg: FormulaPtr);
end
```

837. Constructor.

```
\langle Implementing formula AST 811\rangle +\equiv constructor FlexaryDisjunctiveFormulaObj.Init(const aPos: Position; aLeftArg, aRightArg: FormulaPtr); begin inheritedInit(aPos, aLeftArg, aRightArg); nFormulaSort \leftarrow wsFlexaryDisjunctiveFormula; end;
```

838. Quantified formula. First-order logic is distinguished by the use of terms and quantifying formulas over terms. We have a base class for quantified formulas. Using the Mizar soft type system, quantified variables are "qualified segments".

```
\langle Classes for formula (abstract syntax tree) 810 \rangle + \equiv QuantifiedFormulaPtr = \uparrow QuantifiedFormulaObj; QuantifiedFormulaObj = \mathbf{object} \; (FormulaExpressionObj) nSegment: \; QualifiedSegmentPtr; nScope: FormulaPtr; constructor \; Init(const \; aPos: \; Position; \; aSegment: \; QualifiedSegmentPtr; \; aScope: FormulaPtr); destructor \; Done; \; virtual; end
```

```
839.
      Constructor.
```

```
\langle \text{Implementing formula AST 811} \rangle + \equiv
constructor QuantifiedFormulaObj.Init(const aPos: Position;
                                            aSegment: QualifiedSegmentPtr;
                                            aScope: FormulaPtr);
  begin nFormulaPos \leftarrow aPos; nSegment \leftarrow aSegment; nScope \leftarrow aScope;
  end;
destructor QuantifiedFormulaObj.Done:
  begin dispose(nSegment, Done); dispose(nScope, Done);
  end:
840. Universal formula. When we want to describe a formula of the form "\forall x : T. \varphi[x]" where T is a
soft type and \varphi[x] is a formula parametrized by x.
\langle Classes for formula (abstract syntax tree) 810 \rangle +\equiv
  UniversalFormulaPtr = \uparrow UniversalFormulaObj;
  UniversalFormulaObj = \mathbf{object} (QuantifiedFormulaObj)
    constructor Init(const aPos: Position; aSegment: QualifiedSegmentPtr; aScope: FormulaPtr);
  end
841.
       Constructor.
\langle \text{Implementing formula AST 811} \rangle + \equiv
constructor UniversalFormulaObj.Init(const aPos: Position;
                                            aSegment: QualifiedSegmentPtr;
                                            aScope: FormulaPtr);
  begin inherited\ Init(aPos, aSegment, aScope); nFormulaSort \leftarrow wsUniversalFormula;
  end:
842. Existential formula. The other quantified formula are existentially quantified formulas, which
resemble "\exists x : T. \varphi[x]" where T is a soft type and \varphi[x] is a formula parametrized by x.
\langle Classes for formula (abstract syntax tree) 810 \rangle +\equiv
  ExistentialFormulaPtr = \uparrow ExistentialFormulaObj;
  ExistentialFormulaObj = \mathbf{object} (QuantifiedFormulaObj)
    constructor Init(const aPos: Position; aSegment: QualifiedSegmentPtr; aScope: FormulaPtr);
  end
843. Constructor.
\langle \text{Implementing formula AST 811} \rangle + \equiv
constructor ExistentialFormulaObj.Init(const aPos: Position;
                                             aSegment: QualifiedSegmentPtr;
                                             aScope: FormulaPtr);
  begin inherited Init(aPos, aSegment, aScope); nFormulaSort \leftarrow wsExistentialFormula;
  end;
844. Contradiction formula. The canonical contradiction \perp in Mizar is represented by the reserved
keyword "contradiction".
\langle Classes for formula (abstract syntax tree) 810 \rangle +\equiv
  ContradictionFormulaPtr = \uparrow ContradictionFormulaObj;
  ContradictionFormulaObj = \mathbf{object} (FormulaExpressionObj)
```

constructor *Init* (**const** *aPos*: *Position*);

end

```
845. Constructor.
```

```
\langle Implementing formula AST 811\rangle +\equiv constructor ContradictionFormulaObj.Init(const aPos: Position); begin nFormulaPos \leftarrow aPos; nFormulaSort \leftarrow wsContradiction; end;
```

846. Thesis formula. When we are in the middle of a proof, the goal or obligation left to be proven is called the "thesis".

```
⟨ Classes for formula (abstract syntax tree) 810⟩ +≡

ThesisFormulaPtr = ↑ThesisFormulaObj;

ThesisFormulaObj = object (FormulaExpressionObj)

constructor Init(const aPos: Position);
end
```

847. Constructor.

```
\langle \text{Implementing formula AST 811} \rangle +\equiv  constructor ThesisFormulaObj.Init(\mathbf{const}\ aPos:\ Position); begin nFormulaPos \leftarrow aPos;\ nFormulaSort \leftarrow wsThesis; end;
```

848. Incorrect formula. We also have a node in abstract syntax trees for "incorrect" formulas.

```
⟨ Classes for formula (abstract syntax tree) 810⟩ +≡
IncorrectFormulaPtr = ↑IncorrectFormula;
IncorrectFormula = object (FormulaExpressionObj)
constructor Init(const aPos: Position);
end
```

849. Constructor.

```
\langle Implementing formula AST 811\rangle +\equiv constructor IncorrectFormula.Init(const aPos: Position); begin nFormulaPos \leftarrow aPos; nformulaSort \leftarrow wsErrorFormula; end;
```

File 21

Weakly strict Mizar article

850. The parser "eats in" a mizar article, then produces a .wsx (weakly strict Mizar) XML file containing the abstract syntax tree, and also a .frt article containing the formats for the article.

This strategy should be familiar to anyone who has looked into compilers and interpreters: transform the abstract syntax tree into an intermediate representation, then transform the intermediate representations in various passes.

This module will transform the parse tree to an abstract syntax tree in XML format.

```
⟨ wsmarticle.pas 850⟩ ≡
⟨ GNU License 4⟩
unit wsmarticle;
interface
uses mobjects, errhan, mscanner, syntax, abstract_syntax, xml_dict, xml_inout;
⟨ Publicly declared types in wsmarticle.pas 852⟩
const
⟨ Publicly declared constants in wsmarticle.pas 855⟩
⟨ Publicly declared functions in wsmarticle.pas 853⟩
⟨ Publicly declared functions in wsmarticle.pas 853⟩
⟨ Global variables publicly declared in wsmarticle.pas 1007⟩
implementation
uses mizenv, mconsole, librenv, scanner, xml_parser
mdebug , info end_mdebug;
⟨ Implementation for wsmarticle.pas 854⟩
end .
```

- **851.** Exercise. We will create a class hierarchy for the abstract syntax trees for Mizar. A lot of this is boiler-plate. The reader is invited to write a couple of programs which will:
- (1) read in an EBNF-like grammar and emit the class hierarchy for its abstract syntax tree.
- (2) read in an EBNF-like grammar, and emit the class hierarchy for generating the XML for it.

After all, if you look at the sheer number of sections in this file, it's staggeringly huge. But a lot of it is boiler-plate.

```
852. \langle \text{Publicly declared types in wsmarticle.pas 852} \rangle \equiv
```

```
 \begin{array}{c} \textbf{See also sections } 856,\,862,\,864,\,867,\,868,\,870,\,871,\,875,\,877,\,879,\,881,\,884,\,886,\,888,\,890,\,893,\,895,\,897,\,900,\,906,\,909,\,911,\,913,\\ 915,\,916,\,924,\,926,\,928,\,930,\,946,\,948,\,950,\,952,\,954,\,956,\,958,\,960,\,962,\,964,\,966,\,968,\,970,\,972,\,974,\,976,\,978,\,980,\,982,\\ 984,\,987,\,989,\,991,\,993,\,995,\,997,\,999,\,1001,\,1003,\,1014,\,1097,\,\textbf{and }1148. \end{array}
```

This code is used in section 850.

```
853. \langle \text{Publicly declared functions in wsmarticle.pas 853} \rangle \equiv
```

This code is used in section 850.

Mizar Parser

854. (Implementation for wsmarticle.pas 854) \equiv

See also sections 857, 858, 861, 863, 865, 869, 872, 876, 878, 880, 882, 885, 887, 889, 891, 894, 896, 898, 901, 907, 910, 912, 914, 917, 918, 925, 927, 929, 931, 934, 936, 938, 940, 942, 947, 949, 951, 953, 955, 957, 959, 961, 963, 965, 967, 969, 971, 973, 975, 977, 979, 981, 983, 985, 988, 990, 992, 994, 996, 998, 1000, 1002, 1004, 1005, 1006, 1008, 1013, 1015, 1016, 1017, 1018, 1019, 1020, 1021, 1022, 1023, 1024, 1025, 1026, 1027, 1042, 1043, 1044, 1045, 1058, 1059, 1060, 1061, 1067, 1068, 1069, 1070, 1071, 1072, 1073, 1074, 1075, 1076, 1077, 1078, 1079, 1080, 1095, 1098, 1099, 1100, 1101, 1102, 1103, 1104, 1105, 1106, 1107, 1108, 1109, 1110, 1111, 1112, 1113, 1114, 1115, 1118, 1119, 1120, 1121, 1122, 1123, 1124, 1125, 1126, 1127, 1128, 1129, 1130, 1131, 1132, 1133, 1134, 1135, 1136, 1137, 1138, 1139, 1140, 1141, 1142, 1143, 1144, 1145, 1146, 1147, 1149, 1150, 1151, 1152, 1153, 1154, 1155, 1156, 1157, 1158, 1159, 1160, 1161, 1162, 1163, 1164, 1165, 1166, 1167, 1168, 1169, 1170, 1171, 1172, 1173, 1174, 1175, 1176, 1177, 1178, 1179, 1180, 1181, 1182, 1183, 1184, 1185, 1186, 1187, 1188, 1189, 1190, 1191, and 1192.

This code is used in section 850.

Section 21.1. WEAKLY STRICT TEXT PROPER

855. Mizar provides a grammar for its syntax in the file

/usr/local/doc/mizar/syntax

- It uses a variant of EBNF:
- Terminal symbols are written "in quotes"
- Production rules are separated by vertical lines "|"
- Optional symbols are placed in [brackets]
- Repeated items zero or more times are placed in {braces}.
- Rules end in a period "."

We will freely quote from syntax.txt, rearranging the rules as needed to discuss the relevant parts of Mizar's grammar. We will write the syntax.txt passages in typewriter font.

We should recall the syntax for text items:

```
Text-Proper = Section { Section } .
Section = "begin" { Text-Item } .
Text-Item = Reservation
   Definitional-Item
   Registration-Item
   Notation-Item
   Theorem
   Scheme-Item
   Auxiliary-Item .
Definitional-Item = Definitional-Block ";" .
Registration-Item = Registration-Block ";" .
Theorem = "theorem" Compact-Statement .
Compact-Statement = Proposition Justification ";" .
Justification = Simple-Justification | Proof .
Auxiliary-Item = Statement | Private-Definition .
```

These are the different syntactic classes for "top-level statements" in the text (not the environment header) of a Mizar article. The interested reader can investigate the syntax.txt file more fully to get all the block statements in Mizar. We have already made these different kinds of blocks syntactic values of BlockKind earlier (§692). Now we want to be able to translate them into English. We will just skip ahead and make these different syntactic classes into values of an enumerated type.

```
\langle \text{ Publicly declared constants in wsmarticle.pas } 855 \rangle \equiv
BlockName: array [BlockKind] of string =
  ('Text-Proper', { blMain }
  'Now-Reasoning', { blDiffuse }
  'Hereby-Reasoning', { blHereby }
  'Proof', {blProof}
  Definitional-Block´, { blDefinition }
  'Notation-Block', { blNotation }
  'Case', {blCase}
  'Suppose', { blSuppose }
  'Scheme-Block' { blPublicScheme });
This code is used in section 850.
```

856. Class hierarchy for blocks. We can now translate the grammar for blocks into a class hierarchy. The "text proper" extends an abstract "block" statement. We will provide factory methods "wsTextProper.NewBlock" ■ and "NewItem" for adding a new block (and item) contained within the caller "block". We will be tracking the "kind" of block (§692), and the text proper will need to track which article it belongs to.

All the various kinds of blocks are handled with this one class: proofs, definitions, notations, registrations, cases, suppose blocks, schemes, hereby statements, and so on. However, some of these blocks have extra content which needs their own nodes in the abstract syntax tree, especially Definitions ($\S\S943\ et\ seq.$) and Registrations ($\S\S986\ et\ seq.$).

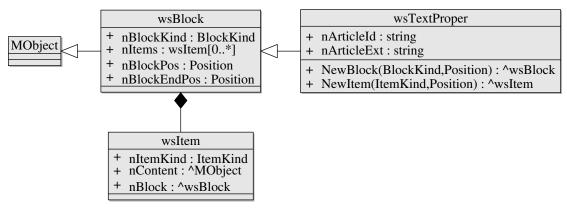


Fig. 4. UML class diagram for wsBlock and related classes.

It is important to stress: wsBlock instances represent all statements which are block statements and all other statements are wsItem instances. Looking back at the different kinds of blocks, you see that they are "block openers" and will expect to have a matching "end" statement closing it.

```
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  wsBlockPtr = \uparrow wsBlock;
  wsBlock = \mathbf{object} \ (MObject)
    nBlockKind: BlockKind;
    nItems: PList; { list of wsItem objects }
    nBlockPos, nBlockEndPos: Position;
    constructor Init(aBlokKind : BlockKind ; const aPos: Position);
    destructor Done; virtual;
  end;
  ⟨ Weakly strict Item class 860⟩;
  wsTextProperPtr = \uparrow wsTextProper;
  wsTextProper = \mathbf{object} (wsBlock)
    nArticleID, nArticleExt: string;
    constructor Init(const aArticleID, aArticleExt: string; const aPos: Position);
    destructor Done; virtual;
    function NewBlock(aBlockKind : BlockKind ; const aPos: Position): wsBlockPtr;
    function NewItem(aItemKind : ItemKind ; const aPos: Position): wsItemPtr;
  end;
```

857. Constructor. We initialize using the inherited wsBlock constructor (§859). The "text proper" refers to a block which is as top-level as possible, so we construct it as a block whose kind is blMain located at aPos.

```
\langle Implementation for wsmarticle.pas 854\rangle +\equiv constructor wsTextProper.Init(const aArticleID, aArticleExt: string; const aPos: Position); begin inheritedInit(blMain, aPos); nArticleID \leftarrow aArticleID; nArticleExt \leftarrow aArticleExt; end; destructor wsTextProper.Done; begin inheritedDone; end;
```

858. Adding statements into a block. we will add a block to a "text proper", which will then construct a block which tracks the caller as its containing block. This requires giving the kind of the newly minted block (§692).

Similarly, when constructing an item which is contained in the block, we need to pass along the item kind (§702).

```
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv function wsTextProper.NewBlock(aBlockKind:BlockKind; const aPos: Position): <math>wsBlockPtr; begin result \leftarrow new(WSBlockPtr,Init(aBlockKind,CurPos)); end; function wsTextProper.NewItem(aItemKind:ItemKind; const aPos: Position): <math>wsItemPtr; begin result \leftarrow new(wsItemPtr,Init(aItemKind,CurPos)); end;
```

859. Block Constructor. Curiously, the *MObject* constructor ($\S187$) is not invoked when constructing a wsBlock. We will also need the position ($\S103$) of the block in the article. The collection of items in the block is initialized to be empty.

```
constructor wsBlock.Init(aBlokKind : BlockKind ; const aPos: Position);
  begin nBlockKind \leftarrow aBlokKind; nBlockPos \leftarrow aPos; nBlockEndPos \leftarrow aPos;
  nItems \leftarrow New(PList, Init(0));
  end;
destructor wsBlock.Done;
  begin dispose (nItems, Done); inherited Done;
  end;
860.
       Text items. An item requires its "kind" (§702) for its syntactic class.
\langle \text{Weakly strict Item class 860} \rangle \equiv
  wsItemPtr = \uparrow wsItem;
  wsItem = \mathbf{object} \ (MObject)
    nItemKind: ItemKind;
    nItemPos, nItemEndPos: Position;
    nContent: PObject;
    nBlock: wsBlockPtr;
    constructor Init(aItemKind : ItemKind ; const aPos: Position);
    destructor Done; virtual;
  end;
```

This code is used in section 856.

```
861. Constructor \langle \text{Implementation for wsmarticle.pas } 854 \rangle +\equiv \text{constructor } wsItem.Init(aItemKind: ItemKind; const } aPos: Position);
begin nItemKind \leftarrow aItemKind; nItemPos \leftarrow aPos; nItemEndPos \leftarrow aPos; nContent \leftarrow nil; nBlock \leftarrow nil;
end;
destructor wsItem.Done;
begin if nBlock \neq nil then dispose(nBlock,Done);
inherited\,Done;
end;
```

862. Pragmas. Mizar supports pragmas (analogous to conditional compilation).

```
⟨ Publicly declared types in wsmarticle.pas 852⟩ +≡
PragmaPtr = ↑PragmaObj;
PragmaObj = object (MObject)
nPragmaStr: string;
constructor Init(aStr: string);
end;
```

863. Constructor.

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv constructor PragmaObj.Init(aStr:string); begin nPragmaStr \leftarrow aStr; end:
```

864. Labels and propositions. A proposition is just a sentence with a label. We will need to represent both of these in our abstract syntax tree.

```
⟨ Publicly declared types in wsmarticle.pas 852⟩ +≡
LabelPtr = ↑LabelObj;
LabelObj = object (MObject)
    nLabelIdNr: integer;
    nLabelPos: Position;
    constructor Init(aLabelId: integer; const aPos: Position);
end;
PropositionPtr = ↑PropositionObj;
PropositionObj = object (mObject)
    nLab: LabelPtr;
    nSntPos: Position;
    nSentence: FormulaPtr;
    constructor Init(aLab: LabelPtr; aSentence: FormulaPtr; const aSntPos: Position);
    destructor Done; virtual;
end;
```

865. Constructor.

```
⟨ Implementation for wsmarticle.pas 854⟩ +≡ constructor LabelObj.Init(aLabelId:integer; const aPos: Position); begin nLabelIdNr \leftarrow aLabelId; nLabelPos \leftarrow aPos; end; constructor PropositionObj.Init(alab:LabelPtr; aSentence: FormulaPtr; const aSntPos: Position); begin nLab \leftarrow aLab; nSntPos \leftarrow aSntPos; nSentence \leftarrow aSentence; end; destructor PropositionObj.Done; begin dispose(nLab,Done); dispose(nSentence,Done); end;
```

866. References. References are either local (i.e., from the file being processed) or library (i.e., from the Mizar math library). The grammar for library references is rather generous. The basic rules are that we have theorem references,

```
\langle article \rangle : \langle number \rangle
```

and definition references,

```
\langle article \rangle: def \langle number \rangle
```

and scheme references,

$$\langle article \rangle$$
: sch $\langle number \rangle$

What makes it tricky is we also allow multiple references from the same article to just add a comma followed by the theorem number

$$\langle article \rangle : \langle number \rangle \{, \langle number \rangle \}$$

or a comma followed by definition numbers

```
\langle article \rangle: def \langle number \rangle \{, def \langle number \rangle \}
```

So far, so good, right? Now we can go even further, mixing theorem references and definitions references from the same article.

We recall the grammar for references:

end;

867. Class structure. We have an abstract "reference" class, which is either a local reference (to a label within the article) or a library reference (to some result in the MML).

869. Constructor. The reference constructors simply populate the appropriate fields in the reference, and the position in the article's text.

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv constructor LocalReferenceObj.Init(aLabId:integer; const aPos: Position); begin nRefSort \leftarrow LocalReference; nLabId \leftarrow aLabId; nRefPos \leftarrow aPosend;
```

constructor *Init*(*aLabId* : *integer* ; **const** *aPos*: *Position*);

870. Library references. This is the abstract class representing either theorem or definition references from an article.

```
⟨ Publicly declared types in wsmarticle.pas 852⟩ +≡
LibraryReferencePtr = ↑LibraryReferenceObj;
LibraryReferenceObj = object (ReferenceObj)
nArticleNr: integer;
end;
```

Theorem and definition references. I am of a divided mind here. On the one hand, we can see that a LibraryReference is a tagged union already, and we do not need separate subclasses for theorem references and definition references. On the other hand, separate subclasses makes things easier when emitting XML for the abstract syntax tree for a Mizar article. Since it is more clear with separate subclasses, and it is better to be clear than clever, I think this design is wiser than the alternatives.

```
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  Theorem Reference Ptr = \uparrow Theorem Reference Obj;
  TheoremReferenceObj = \mathbf{object} \ (LibraryReferenceObj)
    nTheoNr: integer;
    constructor Init(aArticleNr, aTheoNr: integer; const aPos: Position);
  end;
  DefinitionReferencePtr = \uparrow DefinitionReferenceObj;
  DefinitionReferenceObj = \mathbf{object} \ (LibraryReferenceObj)
    nDefNr: integer;
    constructor Init(aArticleNr, aDefNr: integer; const aPos: Position);
  end;
872. Constructor. The reference constructors simply populate the appropriate fields in the reference,
and the position in the article's text.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor TheoremReferenceObj.Init(aArticleNr, aTheoNr: integer;
    const aPos: Position);
  begin nRefSort \leftarrow TheoremReference; nArticleNr \leftarrow aArticleNr; nTheoNr \leftarrow aTheoNr;
  nRefPos \leftarrow aPos
  end;
constructor DefinitionReferenceObj.Init(aArticleNr, aDefNr: integer;
    const aPos: Position);
  begin nRefSort \leftarrow DefinitionReference; <math>nArticleNr \leftarrow aArticleNr; nDefNr \leftarrow aDefNr;
  nRefPos \leftarrow aPos
  end:
      Justifications. The grammar for justifications looks like:
Justification = Simple-Justification
  Proof .
Simple-Justification = Straightforward-Justification
  Scheme-Justification .
Proof = "proof" Reasoning "end" .
Straightforward-Justification = [ "by" References ] .
Scheme-Justification = "from" Scheme-Reference [ "(" References ")" ] .
Proof blocks are already represented as a Block object. We just need to represent the other kinds of
justifications as nodes in the abstract syntax tree.
      The different kinds of inference, since a Justification is a tagged union of sorts.
```

```
\langle \text{Inference kinds (wsmarticle.pas) } 874 \rangle \equiv
  InferenceKind = (infError, infStraightforwardJustification, infSchemeJustification, infProof, infProof)
        infSkippedProof)
```

This code is used in section 867.

destructor Done; virtual;

end;

875. Class structure for justifications. The class hierarchy for justifications reflects the grammar we just discussed. $\langle \text{ Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv$ $JustificationPtr = \uparrow JustificationObj;$ $JustificationObj = \mathbf{object} \ (MObject)$ *nInfSort*: *InferenceKind*; nInfPos: Position;**constructor** *Init*(*aInferSort* : *InferenceKind* ; **const** *aPos*: *Position*); end; 876. Constructor. \langle Implementation for wsmarticle.pas $854 \rangle + \equiv$ **constructor** JustificationObj.Init(aInferSort: InferenceKind; **const** aPos: Position); **begin** $nInfSort \leftarrow aInferSort$; $nInfPos \leftarrow aPos$; end; Simple justifications. These are either "by" a list of references, or "from" a scheme. $\langle \text{ Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv$ $Simple Justification Ptr = \uparrow Simple Justification Obj;$ $Simple Justification Obj = \mathbf{object} (Justification Obj)$ nReferences: PList;**constructor** Init(aInferSort: InferenceKind; **const** aPos: Position); destructor Done; virtual; end; 878. Constructor. $\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv$ **constructor** SimpleJustificationObj.Init(aInferSort : InferenceKind ; **const** aPos: Position); **begin** inherited Init(aInferSort, aPos); $nReferences \leftarrow new(Plist, Init(0))$; end; **destructor** SimpleJustificationObj.Done; **begin** dispose (nReferences, Done); inherited Done; end; Straightforward justification. $\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv$ $StraightforwardJustificationPtr = \uparrow StraightforwardJustificationObj;$ $StraightforwardJustificationObj = \mathbf{object} (SimpleJustificationObj)$ *nLinked*: boolean; nLinkPos: Position;

constructor Init(const aPos: Position; aLinked: boolean; const aLinkPos: Position);

 ${f destructor}\ Scheme Justification Obj. Done;$

begin inherited Done;

end;

880. Constructor. \langle Implementation for wsmarticle.pas $854 \rangle + \equiv$ **constructor** StraightforwardJustificationObj.Init(**const** aPos: Position; *aLinked*: boolean; **const** aLinkPos: Position); **begin** inherited Init (infStraightforwardJustification, aPos); $nLinked \leftarrow aLinked$; $nLinkPos \leftarrow aLinkPos$; end; **destructor** StraightforwardJustificationObj.Done; **begin** inherited Done; end: 881. Scheme justification. $\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv$ $SchemeJustificationPtr = \uparrow SchemeJustificationObj;$ $SchemeJustificationObj = \mathbf{object} (SimpleJustificationObj)$ nSchFileNr: integer; {0 for schemes from current article and positive for library references} nSchemeIdNr: integer; { a number of a scheme for library reference nSchFileNr > 0 or a number of an identifier name for scheme name from current article } nSchemeInfPos: Position; $\textbf{constructor} \ \textit{Init}(\textbf{const} \ \textit{aPos: Position}; \ \textit{aArticleNr}, \textit{aNr: integer});$ **destructor** Done; virtual; end; 882. Constructor. $\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv$ **constructor** SchemeJustificationObj.Init(**const** aPos: Position; aArticleNr, aNr: integer); **begin** inherited Init(infSchemeJustification, aPos); $nSchFileNr \leftarrow aArticleNr$; $nSchemeIdNr \leftarrow aNr$; $nSchemeInfPos \leftarrow aPos;$ end:

286 SCHEMES Mizar Parser §883

Section 21.2. SCHEMES

```
The grammar for schemes looks like:
Scheme-Item = Scheme-Block ";" .
Scheme-Block = "scheme" Scheme-Identifier "{" Scheme-Parameters "}" ":"
  Scheme-Conclusion ["provided" Scheme-Premise {"and" Scheme-Premise}]
  ("proof" | ";") Reasoning "end" .
Scheme-Identifier = Identifier .
Scheme-Parameters = Scheme-Segment "," Scheme-Segment
Scheme-Conclusion = Sentence .
Scheme-Premise = Proposition .
Scheme-Segment = Predicate-Segment | Functor-Segment .
Predicate-Segment =
  Predicate-Identifier {"," Predicate-Identifier} "["[Type-Expression-List] "]" .
Predicate-Identifier = Identifier .
Functor-Segment =
  Functor-Identifier { ", "Functor-Identifier } "(" [Type-Expression-List] ") "Specification .
Functor-Identifier = Identifier .
We begin with the abstract syntax for scheme parameters.
884. Class hierarchy for schemes. We need "predicate segments" and "functor segments" for the
second-order variable parameters to the scheme.
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  SchemeSegmentKind = (PredicateSegment, FunctorSegment);
  SchemeSegmentPtr = \uparrow SchemeSegmentObj;
  SchemeSegmentObj = \mathbf{object} \ (MObject)
    nSegmPos: Position;
    nSeqmSort: SchemeSeqmentKind;
    nVars: PList;
    nTypeExpList: PList;
    constructor Init(const aPos: Position; aSegmSort: SchemeSegmentKind;
      aVars, aTypeExpList: PList);
    destructor Done; virtual;
  end;
885. Constructor.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor SchemeSegmentObj.Init(const aPos: Position;
                                    aSegmSort: SchemeSegmentKind;
                                    a Vars, a TypeExpList: PList);
  \textbf{begin } nSegmPos \leftarrow aPos; \ nSegmSort \leftarrow aSegmSort; \ nVars \leftarrow aVars; \ nTypeExpList \leftarrow aTypeExpList;
destructor SchemeSegmentObj.Done;
  begin dispose(nVars, Done); dispose(nTypeExpList, Done);
  end;
```

§886 Mizar Parser SCHEMES 287

Segment variables for schemes. We need "predicate segments" and "functor segments" for the second-order variable parameters to the scheme. $\langle \text{ Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv$ PredicateSegmentPtr = SchemeSegmentPtr; $FunctorSegmentPtr = \uparrow FunctorSegmentObj;$ $FunctorSegmentObj = \mathbf{object} (SchemeSegmentObj)$ nSpecification: TypePtr;constructor Init(const aPos: Position; aVars, aTypeExpList: PList; aSpecification: TypePtr); destructor Done; virtual; end; Constructor. 887. $\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv$ **constructor** FunctorSegmentObj.Init(**const** aPos: Position; a Vars, a TypeExpList: PList; aSpecification: TypePtr);**begin** inherited $Init(aPos, FunctorSegment, aVars, aTypeExpList); nSpecification <math>\leftarrow$ aSpecification; end; **destructor** FunctorSegmentObj.Done; **begin** dispose (nSpecification, Done); inherited Done; end: Scheme. A Scheme object is the parent class of MSScheme objects in first_identification.pas. But it does not appear to be used anywhere else. This has no place in the abstract syntax tree, for example. $\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv$ $SchemePtr = \uparrow SchemeObj;$ $SchemeObj = \mathbf{object} \ (MObject)$ nSchemeIdNr: integer;nSchemePos: Position; nSchemeParams: PList;nSchemeConclusion: FormulaPtr; nSchemePremises: PList:**constructor** Init(aIdNr: integer; **const** aPos: Position; aParams: PList; aPrems: PList; aConcl: FormulaPtr);destructor Done; virtual; end;

288 SCHEMES Mizar Parser §889

889. Constructor. \langle Implementation for wsmarticle.pas $854 \rangle + \equiv$ **constructor** SchemeObj.Init(aIdNr: integer; **const** aPos: Position; aParams: PList: aPrems: PList; aConcl: FormulaPtr);**begin** $nSchemeIdNr \leftarrow aIdNr$; $nSchemePos \leftarrow aPos$; $nSchemeParams \leftarrow aParams$; $nSchemeConclusion \leftarrow aConcl; nSchemePremises \leftarrow aPrems;$ end: **destructor** SchemeObj.Done; **begin** dispose (nSchemeParams, Done); dispose (nSchemeConclusion, Done); dispose(nSchemePremises, Done);end: 890. Reservations. We can "reserve" an identifier and its type, so we do not need to quantify over it for each theorem. The grammar for it: Reservation = "reserve" Reservation-Segment { "," Reservation-Segment} ";" . Reservation-Segment = Reserved-Identifiers "for" Type-Expression . Reserved-Identifiers = Identifier { "," Identifier } . The data needed for a reserved node in the abstract syntax tree amounts to a list of identifiers and a type. $\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv$ $ReservationSegmentPtr = \uparrow ReservationSegmentObj;$ $ReservationSegmentObj = \mathbf{object} (MObject)$ *nIdentifiers*: *PList*; nResType: TypePtr;**constructor** *Init*(*aIdentifiers* : *PList*; *aType* : *TypePtr*); destructor Done; virtual; end; 891. Constructor. \langle Implementation for wsmarticle.pas $854 \rangle + \equiv$ **constructor** ReservationSegmentObj.Init(aIdentifiers: PList; aType: TypePtr); **begin** $nIdentifiers \leftarrow aIdentifiers; nResType \leftarrow aType;$ end; **destructor** ReservationSegmentObj.Done; **begin** dispose(nIdentifiers, Done); dispose(nResType, Done);

end:

§892 Mizar Parser PRIVATE DEFINITIONS 289

Section 21.3. PRIVATE DEFINITIONS

end;

```
892. The grammar for "private definitions" (which introduces block-local or article-local terms and predicates) looks like:
```

```
Private-Definition = Constant-Definition
   Private-Functor-Definition
  | Private-Predicate-Definition .
Constant-Definition = "set" Equating-List ";" .
Equating-List = Equating {"," Equating }.
Equating = Variable-Identifier "=" Term-Expression .
Private-Functor-Definition = "deffunc" Private-Functor-Pattern "=" Term-Expression ";" .
Private-Predicate-Definition = "defpred" Private-Predicate-Pattern "means" Sentence ";" .
Private-Functor-Pattern = Functor-Identifier "(" [ Type-Expression-List ] ")" .
Private-Predicate-Pattern = Predicate-Identifier "[" [Type-Expression-List ] "]" .
So we really only need to describe private predicates, private functors, and "constant definitions" (which
introduce an abbreviation).
893. Private functors.
\langle Publicly declared types in wsmarticle.pas 852 \rangle + \equiv
  PrivateFunctorDefinitionPtr = \uparrow PrivateFunctorDefinitionObj;
  PrivateFunctorDefinitionObj = \mathbf{object} \ (MObject)
    nFuncId: VariablePtr;
    nTypeExpList: PList;
    nTermExpr: TermPtr:
    constructor Init(aFuncId: VariablePtr; aTypeExpList: Plist; aTerm: TermPtr);
    destructor Done; virtual;
  end;
894. Constructor.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor PrivateFunctorDefinitionObj .Init(aFuncId: VariablePtr; aTypeExpList: Plist;
         aTerm : TermPtr);
  begin nFuncId \leftarrow aFuncId; nTypeExpList \leftarrow aTypeExpList; nTermExpr \leftarrow aTerm;
  end;
destructor PrivateFunctorDefinitionObj.Done;
  begin dispose(nFuncId, Done); dispose(nTypeExpList, Done); dispose(nTermExpr, Done);
  end:
895. Private predicates.
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  PrivatePredicateDefinitionPtr = \uparrow PrivatePredicateDefinitionObj;
  PrivatePredicateDefinitionObj = \mathbf{object} (MObject)
    nPredId: VariablePtr;
    nTupeExpList: PList:
    nSentence \colon Formula Ptr;
    constructor\ Init(aPredId: VariablePtr;\ aTypeExpList: Plist;\ aSnt: FormulaPtr);
    destructor Done; virtual;
```

290 PRIVATE DEFINITIONS Mizar Parser §896

```
896.
       Constructor.
\langle Implementation for wsmarticle.pas 854\rangle + \equiv
\textbf{constructor} \ \textit{PrivatePredicateDefinitionObj}. Init(a\textit{PredId} : \textit{VariablePtr}; \ a\textit{TypeExpList} : \textit{Plist};
         aSnt: FormulaPtr);
  begin nPredId \leftarrow aPredId; nTypeExpList \leftarrow aTypeExpList; nSentence \leftarrow aSnt;
destructor PrivatePredicateDefinitionObj.Done;
  begin dispose(nPredId, Done); dispose(nTypeExpList, Done); dispose(nSentence, Done);
  end;
897.
       Constant definitions. These are little more than abbreviations for terms, and their implementa-
tions reflects this: they are pointers with delusions of grandeur.
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  ConstantDefinitionPtr = \uparrow ConstantDefinitionObj;
  ConstantDefinitionObj = \mathbf{object} \ (MObject)
     nVarId: VariablePtr;
     nTermExpr: TermPtr;
     constructor Init(aVarId : VariablePtr; aTerm : TermPtr);
     destructor Done; virtual;
  end;
898. Constructor.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor ConstantDefinitionObj.Init(aVarId: VariablePtr; aTerm: TermPtr);
  begin nVarId \leftarrow aVarId; nTermExpr \leftarrow aTerm;
  end;
destructor ConstantDefinitionObj.Done;
  begin dispose(nVarId, Done); dispose(nTermExpr, Done);
  end;
```

 $\S 899$ Mizar Parser CHANGING TYPES 291

Section 21.4. CHANGING TYPES

899. Each term has a soft type associated with it, but we can "reconsider" or change its type. Mizar requires a proof that the term really has the new type. The grammar for this statement:

```
Type-Changing-Statement =
   "reconsider" Type-Change-List "as" Type-ExpressionSimple-Justification ";" .
Type-Change-List =
   (Equating | Variable-Identifier) {"," (Equating | Variable-Identifier)} .
```

This requires a bit of work since we really have *two* types of reconsiderations within a single reconsider statement:

```
(1) "reconsider \langle identifier \rangle as \langle type \rangle"
(2) "reconsider \langle identifier \rangle = \langle term \rangle as \langle type \rangle"
```

The trick is to represent a Type-Change-List as a list of Type-Changes. Then a Type-Change-Statement is just a Type-Change-List and a type.

900. Class hierarchy.

```
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  TypeChangeSort = (Equating, VariableIdentifier);
  TypeChangePtr = \uparrow TypeChangeObj;
  TypeChangeObj = \mathbf{object} \ (MObject)
     nTypeChangeKind: TypeChangeSort;
     nVar: VariablePtr;
     nTermExpr: TermPtr:
     constructor Init(aKind: TypeChangeSort; aVar: VariablePtr; aTerm: TermPtr);
     destructor Done; virtual;
  end;
  ⟨Example classes (wsmarticle.pas) 903⟩
  TypeChangingStatementPtr = \uparrow TypeChangingStatementObj:
  TypeChangingStatementObj = \mathbf{object} (MObject)
     nTypeChangeList: PList;
     nTypeExpr: TypePtr;
     nJustification: SimpleJustificationPtr;
     \textbf{constructor} \ \textit{Init} (\textit{aTypeChangeList}: \textit{PList}; \ \textit{aTypeExpr}: \textit{TypePtr};
               aJustification: SimpleJustificationPtr);
     destructor Done; virtual;
  end;
```

292 CHANGING TYPES Mizar Parser §901

901. Constructor.

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor TypeChangeObj.Init(aKind : TypeChangeSort; aVar : VariablePtr; aTerm : TermPtr);
  begin nTypeChangeKind \leftarrow aKind; nVar \leftarrow aVar; nTermExpr \leftarrow aTerm;
  end;
{\bf destructor}\ \textit{TypeChangeObj.Done};
  begin dispose(nVar, Done);
  if nTermExpr \neq nil then dispose(nTermExpr, Done);
  end;
⟨ Constructors for example statements (wsmarticle.pas) 904⟩
{f constructor}\ TypeChanqingStatementObj.Init(aTypeChanqeList:PList;\ aTypeExpr:TypePtr;
         a Justification: Simple Justification Ptr);
  begin nTypeChangeList \leftarrow aTypeChangeList; <math>nTypeExpr \leftarrow aTypeExpr;
  nJustification \leftarrow aJustification;
  end;
{\bf destructor}\ \textit{Type Changing Statement Obj. Done};
  begin dispose(nTypeChangeList, Done); dispose(nTypeExpr, Done); dispose(nJustification, Done);
  end;
```

 $\S902$ Mizar Parser PROOF STEPS 293

Section 21.5. PROOF STEPS

Exemplification = "take" Example {"," Example} ";" .

902. Most of the proof steps are handled in generic text-item objects. But there are a few which are outside that tagged union. In particular: existential elimination (consider $\langle variables \rangle$ such that $\langle formula \rangle$), existential introduction (take $\langle terms \rangle$), and concluding statements (thus $\langle formula \rangle$).

903. Examples, existential introduction. The proof step "take x" transforms goals of the form $\exists x. P[x]$ into a new goal P[x]. The grammar for examples looks like:

```
Example = Term-Expression | Variable-Identifier "=" Term-Expression .
\langle \text{Example classes (wsmarticle.pas) } 903 \rangle \equiv
  ExamplePtr = \uparrow ExampleObj;
  ExampleObj = \mathbf{object} \ (MObject)
    nVarId: VariablePtr;
    nTermExpr: TermPtr;
    constructor Init(aVarId : VariablePtr; aTerm : TermPtr);
    destructor Done; virtual;
  end;
This code is used in section 900.
904. Constructor.
\langle \text{Constructors for example statements (wsmarticle.pas) } 904 \rangle \equiv
constructor Example Obj. Init (a VarId: Variable Ptr; a Term: TermPtr);
  begin nVarId \leftarrow aVarId; nTermExpr \leftarrow aTerm;
  end;
destructor ExampleObj.Done;
  begin if nVarId \neq nil then dispose(nVarId, Done);
  if nTermExpr \neq nil then dispose(nTermExpr, Done);
This code is used in section 901.
905. Existential elimination. We continue plugging along with the statements, and existential elimination.
nation (or "choice") statements are the next one.
Linkable-Statement = Compact-Statement
    Choice-Statement
    Type-Changing-Statement
   Iterative-Equality .
Choice-Statement = "consider" Qualified-Variables "such" ConditionsSimple-Justification ";" .
       \langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  ChoiceStatementPtr = \uparrow ChoiceStatementObj;
  ChoiceStatementObj = \mathbf{object} \ (MObject)
    nQualVars: PList;
    nConditions: PList;
    nJustification: SimpleJustificationPtr;
    constructor\ Init(aQualVars, aConds: PList;\ aJustification: SimpleJustificationPtr);
    destructor Done; virtual;
  end;
```

294 PROOF STEPS Mizar Parser §907

```
907.
      Constructor.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor\ ChoiceStatementObj.Init(aQualVars, aConds: PList;\ aJustification: SimpleJustificationPtr);
  begin nQualVars \leftarrow aQualVars; nConditions \leftarrow aConds; nJustification \leftarrow aJustification;
  end:
destructor ChoiceStatementObj.Done;
  \mathbf{begin}\ dispose(nQualVars,Done);\ dispose(nConditions,Done);\ dispose(nJustification,Done);
  end:
908. Conclusion statements. We recall the grammar for conclusion statements:
Conclusion = ( "thus" | "hence" ) ( Compact-Statement | Iterative-Equality )
  Diffuse-Conclusion .
Diffuse-Conclusion = "thus" Diffuse-Statement | "hereby" Reasoning "end" ";" .
Iterative-Equality =
[ Label-Identifier ":" ] Term-Expression "=" Term-ExpressionSimple-Justification
                                               ".=" Term-Expression Simple-Justification
                                             { ".=" Term-Expression Simple-Justification } ";" .
NOTE: the whitespace in the Iterative-Equality rule is unimportant, but that is how Mizar users often
structure them (to align the equals sign).
909. Abstract base class.
\langle Publicly declared types in wsmarticle.pas 852 \rangle + \equiv
  Regular Statement Kind = (st Diffuse Statement, st Compact Statement, st Iterative Equality);
  RegularStatementPtr = \uparrow RegularStatementObj;
  RegularStatementObj = \mathbf{object} \ (MObject)
    nStatementSort: RegularStatementKind;
    nLab: LabelPtr;
    constructor Init(aStatementSort : RegularStatementKind);
    destructor Done; virtual;
  end;
910. Constructor.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor RegularStatementObj.Init(aStatementSort : RegularStatementKind);
  begin nStatementSort \leftarrow aStatementSort;
  end:
destructor RegularStatementObj.Done;
  begin inherited Done;
  end;
911. Thus statement. The conclusion of a proof (idiomatically "thus thesis") is always a "thus",
which Mizar calls a "diffuse statement".
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  DiffuseStatementPtr = \uparrow DiffuseStatementObj;
  DiffuseStatementObj = \mathbf{object} \ (RegularStatementObj)
    constructor Init(aLab : LabelPtr; aStatementSort : RegularStatementKind);
    destructor Done; virtual;
  end;
```

 $\S912$ Mizar Parser PROOF STEPS 295

```
912.
       Constructor.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor DiffuseStatementObj.Init(aLab: LabelPtr; aStatementSort: RegularStatementKind);
  begin inherited Init(stDiffuseStatement); nLab \leftarrow aLab; nStatementSort \leftarrow aStatementSort;
  end:
destructor DiffuseStatementObj.Done;
  begin dispose(nLab, Done);
  end;
      Compact statements. We recall the syntax for a compact statement is:
Compact-Statement = Proposition Justification ";" .
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  CompactStatementPtr = \uparrow CompactStatementObj;
  CompactStatementObj = \mathbf{object} \ (RegularStatementObj)
    nProp: PropositionPtr;
    nJustification: JustificationPtr;
    constructor\ Init(aProp: PropositionPtr;\ aJustification: JustificationPtr);
    destructor Done; virtual;
  end;
914. Constructor.
\langle Implementation for wsmarticle.pas 854\rangle +=
constructor \ CompactStatementObj.Init(aProp: PropositionPtr; aJustification: JustificationPtr);
  begin inherited Init(stCompactStatement); nProp \leftarrow aProp; nJustification \leftarrow aJustification;
  end:
destructor CompactStatementObj.Done;
  begin if nJustification \neq nil then <math>dispose(nJustification, Done);
  inherited Done;
  end:
915. Iterative equality. Chain of equations, where we keep transforming the right-hand side until we
arrive at the desired outcome.
\langle \text{ Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  IterativeStepPtr = \uparrow IterativeStepObj;
  IterativeStepObj = \mathbf{object} (MObject)
    nIterPos: Position;
    nTerm: TermPtr;
    nJustification: SimpleJustificationPtr;
    constructor Init(const aPos: Position; aTerm: TermPtr; aJustification: JustificationPtr);
    destructor Done; virtual;
  end;
916. (Publicly declared types in wsmarticle.pas 852) +\equiv
  IterativeEqualityPtr = \uparrow IterativeEqualityObj;
  IterativeEqualityObj = \mathbf{object} (CompactStatementObj)
    nIterSteps: PList;
    constructor Init(aProp: PropositionPtr; aJustification: JustificationPtr; aIters: PList);
    destructor Done; virtual;
```

end;

296 PROOF STEPS Mizar Parser $\S917$

917. Constructor.

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor IterativeStepObj.Init(const aPos: Position; aTerm: TermPtr; aJustification:
          JustificationPtr);
  begin nIterPos \leftarrow aPos; nTerm \leftarrow aTerm; nJustification \leftarrow SimpleJustificationPtr(aJustification);
destructor IterativeStepObj.Done;
  begin dispose(nTerm, Done); dispose(nJustification, Done);
  end;
918. Constructor.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
{f constructor}\ Iterative Equality Obj. Init (a Prop: Proposition Ptr;\ a Justification: Justification Ptr;
          aIters: PList);
  begin inherited Init(aProp, aJustification); nStatementSort \leftarrow stIterativeEquality; nIterSteps \leftarrow aIters;
  end;
destructor IterativeEqualityObj.Done;
  begin dispose (nIterSteps, Done); inherited Done;
  end;
```

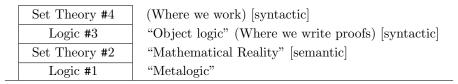
919. Remaining proof steps? So where are the other proof steps like let or assume? Well, these are handled as "generic text items" and use the *TextItem* class (§860).

§920 Mizar Parser STRUCTURES 297

Section 21.6. STRUCTURES

920. Just an aside first on "what is a structure in Mathematics?" Logic textbooks assume an *intuitive* (i.e., not formal) "finitary metatheory" following Hilbert and his famous Programme in the foundations of Mathematics. We will build a "skyscraper" atop this foundation of finitary metatheory. The first thing we do is describe a logic, the first floor in our sky scraper. This "Logic #1" is the metalogic we use to construct an axiomatic set theory, "Set Theory #2". We use "Set Theory #2" to construct another floor, a "Logic #3", which then builds another floor "Set Theory #4", and so on. We can potentially iterate building as many floors as we want, but 4 is sufficient for our purposes.

We assert that "Set Theory #2" is the Platonic "mathematical reality". Then "Logic #3" is the (ambient) logic we use to do Mathematics; it is purely "syntactic", a language for expressing proofs and definitions. Mizar's proof steps, formulas, and definitions corresponds to "Logic #3". With it, we describe an axiomatic "Set Theory #4", which is Tarski–Grothendieck set theory for Mizar. Sketching this situation out diagrammatically:



Finitary Metatheory

Fig. 5. Mathematical Platonism as a skyscaper.

Now, "mathematical objects" live in "Set Theory #2". Model theory studies structures (objects in "Set Theory #2") of theories (described in "Logic #3"). Since we "believe" that set theory "describes reality", that means we just need to describe ["syntactic"] theories using "Set Theory #4" and their "real world occurrences" in "Set Theory #2". (Well, this is a gloss, model theory sets up two additional floors in the skyscraper, and studies "models" of theories described using Logic #5 and Set Theory #6 in Set Theory #4— and we pretend it describes the relationship between Set Theory #2 and the "syntactic floors" of the Mathematical skyscraper.)

How do we *syntactically* describe these "structures"? Well, we *know* they are not "first-class citizens" in Mizar, in the sense that they are not "just" a tuple. How do we know this? Gilbert Lee and Piotr Rudnicki's "Alternative Aggregates in Mizar" (in *MKM 2007*, Springer, pp.327–341; doi:10.1007/978-3-540-73086-6_26) discuss how to implement first-class structures in Mizar. This means that *technically* structures live in Logic #3. Field symbols are terms in Logic #3.

921. Why do we need this convoluted skyscraper? Without it, how do we describe a "true" formula? We can only speak of a provable formula. Bourbaki's Theory of Sets (I $\S 2.2$) confuses "provable" with "true" formulas (they speak of a formula being "false in a theory \mathcal{T} " as being synonymous with the formula contradicting the axioms for a theory, and true in a theory as being synonymous for being a logical consequence from the axioms for a theory). This only matters for Mathematical Platonists. Formalists (like the author) would find this discussion muddled and nearly metaphysical, generating more heat than light.

298 STRUCTURES Mizar Parser $\S922$

922. Aside: finitary metatheory, programming languages, implementing proof assistants.

How does that diagram in Figure 5 of the last section compare to the *actual implementation* of Mizar? Well, a proof assistant replaces the "finitary metatheory" with an actual programming language. Then, since only Mathematical Platonists care about the "Metalogic" and "Mathematical reality", we jump ahead to implement Logic #3 — this is what happens in Mizar and other proof assistants: we implement a "purely formal" (purely syntactic) logic using a programming language. Curiously, this reflects Bourbaki's approach to the foundations of Mathematics.

We should note that programming languages are strictly stronger than finitary metatheory, since programming languages are *Turing complete*. This means they support general recursion, whereas finitary metatheory supports only primitive recursive functions. For an example of a "programming language" which is equally as strong as a finitary metatheory, see Albert R. Meyer and Dennis M. Ritchie, "The complexity of loop programs" (*ACM '67 Proc.*, 1967, doi:10.1145/800196.806014).

Is Turing completeness "too much" for a finitary metatheory? The short answer is: yes. Even restricting a Turing complete programming language is "too much" to be finitary. Gödel's System T was developed to preserve the "constructive character" while jettisoning the "finitary character" of Hilbert's finitary metatheory, and System T is not even Turing complete. See Kurt Gödel's Collected Works (vol. II, Oxford University Press, doi:10.1093/oso/9780195147216.001.0001, 1989; viz., pp. 245–247) for his discussion of System T. The interested reader should consult David A. Turner's "Elementary strong functional programming" (in Int. Symp. on Funct. Program. Lang. in Educ., eds P.H. Hartel and R. Plasmeijer, Springer, pages 1–13, doi:10.1007/3-540-60675-0_35) for how to obtain System T by restricting any statically typed functional programming language.

923. Grammar for structures. We can recall the syntax for structures and fields:

```
Structure-Definition =
   "struct" [ "(" Ancestors ")" ] Structure-Symbol [ "over" Loci ] "(#" Fields "#)" ";" .
Ancestors = Structure-Type-Expression { "," Structure-Type-Expression } .
Structure-Symbol = Symbol .
Loci = Locus { "," Locus } .
Fields = Field-Segment { "," Field-Segment } .
Locus = Variable-Identifier .
Variable-Identifier = Identifier .
Field-Segment = Selector-Symbol { "," Selector-Symbol } Specification .
Selector-Symbol = Symbol .
```

924. Field symbol. A "field symbol" refers to the identifier used for a field in a structure, but not its type.

```
⟨ Publicly declared types in wsmarticle.pas 852⟩ +≡
FieldSymbolPtr = ↑FieldSymbolObj;
FieldSymbolObj = object (MObject)
    nFieldPos: Position;
    nFieldSymbol: integer;
    constructor Init(const aPos: Position; aFieldSymbNr: integer);
end;
```

925. Constructor.

```
\langle Implementation for wsmarticle.pas 854\rangle +\equiv constructor FieldSymbolObj.Init(const aPos: Position; aFieldSymbNr: integer); begin nFieldPos \leftarrow aPos; nFieldSymbol \leftarrow aFieldSymbNr; end;
```

§926 Mizar Parser STRUCTURES 299

926. **Field segment.** A field segment refers to a list of 1 or more field symbols, and the associated type it has. $\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv$ $FieldSegmentPtr = \uparrow FieldSegmentObj;$ $FieldSegmentObj = \mathbf{object} \ (MObject)$ nFieldSegmPos: Position;nFields: PList;nSpecification: TypePtr;**constructor** *Init*(**const** *aPos*: *Position*; *aFields*: *PList*; *aSpec*: *TypePtr*); **destructor** Done; virtual; end; 927. Constructor. \langle Implementation for wsmarticle.pas 854 $\rangle +=$ **constructor** FieldSegmentObj.Init(**const** aPos: Position; aFields: PList; aSpec: TypePtr); **begin** $nFieldSegmPos \leftarrow aPos; nFields \leftarrow aFields; nSpecification \leftarrow aSpec;$ end; **destructor** FieldSegmentObj.Done; **begin** dispose(nFields, Done); dispose(nSpecification, Done); end; **928.** Locus. A "locus" refers to a term or type parametrizing a definition. $\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv$ $LocusPtr = \uparrow LocusObj;$ $LocusObj = \mathbf{object} \ (MObject)$ nVarId: integer;nVarIdPos: Position;**constructor** *Init*(**const** *aPos*: *Position*; *aIdentNr*: *integer*); end; 929. Constructor. \langle Implementation for wsmarticle.pas 854 $\rangle + \equiv$ constructor LocusObj.Init(const aPos: Position; aIdentNr: integer); **begin** $nVarId \leftarrow aIdentNr; nVarIdPos \leftarrow aPos;$ end;

300 STRUCTURES Mizar Parser $\S 930$

930. Structure definition. Finally, structures are finite maps from selectors to terms, with structure inheritance thrown into the mix. They may be defined "over" a finite list of types (e.g., a module structure is "over" a ring). Note that we need to first introduce "patterns" before describing the structure definition, since "patterns" are needed in definitions.

```
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  ⟨Pattern objects (wsmarticle.pas) 933⟩
  StructureDefinitionPtr = \uparrow StructureDefinitionObj;
  StructureDefinitionObj = \mathbf{object} (MObject)
    nStrPos: Position;
    nAncestors: PList;
    nDefStructPattern: ModePatternPtr;
    nSgmFields: PList;
    constructor Init(const aPos: Position; aAncestors: PList; aStructSymb: integer;
       aOverArgs: PList; aFields: PList);
    destructor Done; virtual;
  end;
931. Constructor.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor StructureDefinitionObj.Init(const aPos: Position; aAncestors: PList;
                                            aStructSymb: integer; aOverArgs: PList; aFields: PList);
  begin nStrPos \leftarrow aPos; nAncestors \leftarrow aAncestors;
  nDefStructPattern \leftarrow new(ModePatternPtr, Init(aPos, aStructSymb, aOverArgs));
  nDefStructPattern \uparrow . nPatternSort \leftarrow itDefStruct; \ nSgmFields \leftarrow aFields;
destructor StructureDefinitionObj.Done;
  begin dispose (nAncestors, Done); dispose (nDefStructPattern, Done); dispose (nSqmFields, Done);
  end;
```

 $\S932$ Mizar Parser PATTERNS 301

Section 21.7. PATTERNS

```
932. A "Pattern" in Mizar is a format with the type information for all the arguments around a term.
The notion of a "Pattern" also refers to the definiendum of a definition. The syntax of patterns
Mode-Pattern = Mode-Symbol [ "of" Loci ] .
Attribute-Pattern = Locus "is" [ Attribute-Loci ] Attribute-Symbol .
Attribute-Loci = Loci | "(" Loci ")" .
Predicate-Pattern = [ Loci ] Predicate-Symbol [ Loci ] .
Functor-Pattern = [ Functor-Loci ] Functor-Symbol [ Functor-Loci ]
  | Left-Functor-Bracket Loci Right-Functor-Bracket .
Functor-Loci = Locus | "(" Loci ")" .
933. Base class for patterns.
\langle Pattern objects (wsmarticle.pas) 933 \rangle \equiv
  PatternPtr = \uparrow PatternObj;
  PatternObj = \mathbf{object} \ (mObject)
    nPatternPos: Position;
    nPatternSort: ItemKind;
    constructor Init(const aPos: Position; aSort: ItemKind);
  end;
See also sections 935, 937, 939, and 941.
This code is used in section 930.
934. (Implementation for wsmarticle.pas 854) +\equiv
constructor PatternObj.Init(const aPos: Position; aSort: ItemKind);
  begin nPatternPos \leftarrow aPos; nPatternSort \leftarrow aSort;
  end;
      Mode patterns. The syntax for "mode patterns" looks like:
Mode-Pattern = Mode-Symbol [ "of" Loci ] .
⟨Pattern objects (wsmarticle.pas) 933⟩ +≡
  ModePatternPtr = \uparrow ModePatternObj;
  ModePatternObj = \mathbf{object} \ (PatternObj)
    nModeSymbol: Integer;
    nArgs: PList;
    constructor Init(const aPos: Position; aSymb: integer; aArgs: PList);
    destructor Done; virtual;
  end;
936. Constructor.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor ModePatternObj.Init(const aPos: Position; aSymb: integer; aArgs: PList);
  begin inherited Init(aPos, itDefMode); nModeSymbol \leftarrow aSymb; nArgs \leftarrow aArgs;
  end;
destructor ModePatternObj.Done;
  begin dispose(nArgs, Done);
  end:
```

302 PATTERNS Mizar Parser §937

937. Attribute patterns. Attributes can have loci prefixing the attribute symbol, but *not* suffixing the attribute symbol.

```
Attribute-Pattern = Locus "is" [ Attribute-Loci ] Attribute-Symbol .
Attribute-Loci = Loci | "(" Loci ")" .
⟨Pattern objects (wsmarticle.pas) 933⟩ +≡
  AttributePatternPtr = \uparrow AttributePatternObj;
  AttributePatternObj = \mathbf{object} (PatternObj)
    nAttrSymbol: Integer;
    nArg: LocusPtr;
    nArgs: PList;
    constructor Init(const aPos: Position; aArq: LocusPtr; aSymb: integer; aArqs: PList);
    destructor Done; virtual;
  end;
938.
      Constructor.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor AttributePatternObj.Init(const aPos: Position; aArq: LocusPtr; aSymb: integer; aArqs:
         PList):
  begin inherited Init(aPos, itDefAttr); nAttrSymbol \leftarrow aSymb; nArg \leftarrow aArg; nArgs \leftarrow aArgs;
  end;
destructor AttributePatternObj.Done;
  begin dispose(nArg, Done); dispose(nArgs, Done);
  end:
939. Predicate patterns. Predicates can have loci on either side of the predicate symbol, without
requiring parentheses (unlike functors).
Predicate-Pattern = [ Loci ] Predicate-Symbol [ Loci ] .
⟨Pattern objects (wsmarticle.pas) 933⟩ +≡
  PredicatePatternPtr = \uparrow PredicatePatternObj;
  PredicatePatternObj = \mathbf{object} (PatternObj)
    nPredSymbol: Integer;
    nLeftArgs, nRightArgs: PList;
    constructor Init(const aPos: Position; aLArgs: PList; aSymb: integer; aRArgs: PList);
    destructor Done; virtual;
  end;
940. Constructor.
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
constructor PredicatePatternObj.Init(const aPos: Position;
       aLArgs: PList; aSymb: integer; aRArgs: PList);
  begin inherited Init(aPos, itDefPred); nPredSymbol \leftarrow aSymb; nLeftArgs \leftarrow aLArgs;
  nRightArgs \leftarrow aRArgs;
  end;
destructor PredicatePatternObj.Done;
  begin dispose(nLeftArgs, Done); dispose(nRightArgs, Done);
  end;
```

 $\S941$ Mizar Parser PATTERNS 303

941. Functor pattern. Functors can have loci on either side. If more than one locus is used on one side, then it must be placed in parentheses and comma-separated. The syntax:

```
Functor-Pattern = [ Functor-Loci ] Functor-Symbol [ Functor-Loci ]
  | Left-Functor-Bracket Loci Right-Functor-Bracket .
Functor-Loci = Locus | "(" Loci ")" .
⟨Pattern objects (wsmarticle.pas) 933⟩ +≡
  FunctorSort = (InfixFunctor, CircumfixFunctor);
  FunctorPatternPtr = \uparrow FunctorPatternObj;
  FunctorPatternObj = \mathbf{object} (PatternObj)
    nFunctKind: FunctorSort;
    constructor Init(const aPos: Position; aKind: FunctorSort);
  end;
  CircumfixFunctorPatternPtr = \uparrow CircumfixFunctorPatternObj;
  CircumfixFunctorPatternObj = \mathbf{object} (FunctorPatternObj)
    nLeftBracketSymb, nRightBracketSymb: integer;
    nArgs: PList;
    constructor Init(const aPos: Position; aLBSymb, aRBSymb: integer; aArgs: PList);
    destructor Done; virtual;
  end:
  InfixFunctorPatternPtr = \uparrow InfixFunctorPatternObj;
  InfixFunctorPatternObj = \mathbf{object} (FunctorPatternObj)
    nOperSymb: integer;
    nLeftArgs, nRightArgs: PList;
    constructor Init(const aPos: Position; aLArgs: PList; aSymb: integer; aRArgs: PList);
    destructor Done; virtual;
  end;
942. Constructor.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor FunctorPatternObj.Init(const aPos: Position; aKind: FunctorSort);
  begin inherited Init(aPos, itDefFunc); nFunctKind \leftarrow aKind;
  end;
constructor CircumfixFunctorPatternObj.Init(const aPos: Position; aLBSymb, aRBSymb: integer;
         aArgs: PList);
  begin inherited Init(aPos, CircumfixFunctor); nLeftBracketSymb \leftarrow aLBSymb;
  nRightBracketSymb \leftarrow aRBSymb; nArgs \leftarrow aArgs;
  end:
destructor CircumfixFunctorPatternObj.Done;
  begin dispose(nArgs, Done);
  end:
constructor InfixFunctorPatternObj.Init(const aPos: Position; aLArgs: PList; aSymb: integer; aRArgs:
  begin inherited\ Init(aPos, InfixFunctor);\ nOperSymb \leftarrow aSymb;\ nLeftArgs \leftarrow aLArgs;
  nRightArgs \leftarrow aRArgs;
  end:
destructor InfixFunctorPatternObj.Done;
  begin dispose(nLeftArgs, Done); dispose(nRightArgs, Done);
  end;
```

304 DEFINITIONS Mizar Parser §943

Section 21.8. DEFINITIONS

943. In Mizar, we can redefine an existing definition (either changing the type of a term or "the right hand side" of a definition) or we can introduce a new definition. There are 5 different things we can introduce: structures, modes [types], functors [terms], predicates, and attributes. Rather than bombard the reader with a long chunk of grammar, let us divide it up into easy-to-digest pieces. The basic block structure of a definition is the same for all these situations, its grammar looks like:

```
Definitional-Item = Definitional-Block ";" .

Definitional-Block = "definition" { Definition-Item | Definition | Redefinition } "end" .

Definition-Item = Loci-Declaration | Permissive-Assumption | Auxiliary-Item .

Loci-Declaration = "let" Qualified-Variables [ "such" Conditions ] ";" .

Permissive-Assumption = Assumption .

Definition = Structure-Definition | Mode-Definition | Functor-Definition | Predicate-Definition | Predicate-Definition | Attribute-Definition .
```

944. Redefinitions. Redefinitions allow us to alter the type or meaning of a definition. This isn't willy-nilly, the user still needs to prove the redefined version is logically equivalent to the initial definition.

```
Redefinition = "redefine" ( Mode-Definition | Functor-Definition | Predicate-Definition | Attribute-Definition ) .
```

945. Structure definitions. Structures intuitively correspond to new "gadgets" (sets equipped with extra structure), which is often presented in Mathematics as "just another tuple". Mizar allows structures to inherit other structures, so a topological group extends a topological space structure and a magma structure (since a group in Mizar is a magma with some extra properties).

```
Structure-Definition =
  "struct" [ "(" Ancestors ")" ] Structure-Symbol [ "over" Loci ] "(#" Fields "#)" ";" .
Ancestors = Structure-Type-Expression { "," Structure-Type-Expression } .
Structure-Symbol = Symbol .
Loci = Locus { "," Locus } .
Fields = Field-Segment { "," Field-Segment } .
Locus = Variable-Identifier .
Variable-Identifier = Identifier .
Field-Segment = Selector-Symbol { "," Selector-Symbol } Specification .
Selector-Symbol = Symbol .
Specification = "->" Type-Expression .
946. Definiens. Recall the grammar for Definiens looks like:
Definiens = Simple-Definiens | Conditional-Definiens .
Simple-Definiens = [ ":" Label-Identifier ":" ] ( Sentence | Term-Expression ) .
Label-Identifier = Identifier .
Conditional-Definiens = [ ":" Label-Identifier ":" ] Partial-Definiens-List
  [ "otherwise" ( Sentence | Term-Expression ) ] .
Partial-Definiens-List = Partial-Definiens { "," Partial-Definiens } .
Partial-Definiens = ( Sentence | Term-Expression ) "if" Sentence .
```

§946 Mizar Parser DEFINITIONS 305

We begin with a base class for definiens. This is extended by SimpleDefiniens and ConditionalDefiniens classes.

```
\langle \text{ Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  How To Define = (df Empty, df Means, df Equals);
  DefiniensSort = (SimpleDefiniens, ConditionalDefiniens);
  DefiniensPtr = \uparrow DefiniensObj;
  DefiniensObj = \mathbf{object} \ (MObject)
    nDefSort: DefiniensSort;
    nDefPos: Position;
    nDefLabel: LabelPtr;
    constructor Init(const aPos: Position; aLab: LabelPtr; aKind: DefiniensSort);
    destructor Done: virtual:
  end;
947. Constructor.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor DefiniersObj.Init(const aPos: Position; aLab: LabelPtr; aKind: DefiniersSort);
  begin nDefSort \leftarrow aKind; nDefPos \leftarrow aPos; nDefLabel \leftarrow aLab;
destructor DefiniensObj.Done;
  begin if nDefLabel \neq nil then dispose(nDefLabel, Done);
  end;
948. Definiens expression. These nodes in the abstract syntax tree describe "the right hand side" of a
definition. A simple definiens is just a pointer to one definiens expression object, for example.
\langle \text{ Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  DefExpressionPtr = \uparrow DefExpressionObj;
  DefExpressionObj = \mathbf{object} \ (MObject)
    nExprKind: ExpKind;
    nExpr: PObject;
    constructor Init(aKind : ExpKind; aExpr : PObject);
    destructor Done; virtual;
  end;
949. Constructor.
\langle Implementation for wsmarticle.pas 854\rangle + \equiv
constructor DefExpressionObj.Init(aKind: ExpKind; aExpr: Pobject);
  begin nExprKind \leftarrow aKind; nExpr \leftarrow aExpr;
  end:
destructor DefExpressionObj.Done;
  begin dispose(nExpr, Done);
  end;
```

306 DEFINITIONS Mizar Parser §950

```
Simple definiens. This is the "default" definiens, i.e., the definiens which are not "by cases".
\langle \text{ Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  SimpleDefiniensPtr = \uparrow SimpleDefiniensObj;
  SimpleDefiniensObj = \mathbf{object} (DefiniensObj)
    nExpression: DefExpressionPtr;
    constructor Init(const aPos: Position; aLab: LabelPtr; aDef: DefExpressionPtr);
    destructor Done; virtual;
  end;
951. Constructor.
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
constructor SimpleDefiniensObj.Init(const aPos: Position; aLab: LabelPtr; aDef: DefExpressionPtr);
  begin inherited Init(aPos, aLab, SimpleDefiniens); nExpression <math>\leftarrow aDef;
  end:
destructor SimpleDefiniensObj.Done;
  begin dispose (nExpression, Done); inherited Done;
  end;
952. Definition for particular case. We have "(sentence or term) if (quard condition)" represented
by a couple of pointers: one to the "sentence or term" definiens, and the second to the "guard" condition.
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  PartDefPtr = \uparrow PartDefObj;
  PartDefObj = \mathbf{object} \ (MObject)
    nPartDefiniens: DefExpressionPtr;
    nGuard: FormulaPtr;
    constructor Init(aPartDef : DefExpressionPtr; aGuard : FormulaPtr);
    destructor Done; virtual;
  end:
953. Constructor.
\langle Implementation for wsmarticle.pas 854\rangle + \equiv
constructor PartDefObj.Init(aPartDef: DefExpressionPtr; aGuard: FormulaPtr);
  begin nGuard \leftarrow aGuard; nPartDefiniens \leftarrow aPartDef;
  end;
destructor PartDefObj.Done;
  begin dispose (nPartDefiniens, Done); dispose (nGuard, Done);
  end;
954. Conditional definiens. A conditional definiens consists of a finite list of pointers to PartDef
objects, and a pointer to the default "otherwise" definien.
⟨Publicly declared types in wsmarticle.pas 852⟩ +≡
  Conditional Definiens Ptr = \uparrow Conditional Definiens Obj;
  Conditional Definiens Obj = \mathbf{object} \ (Definiens Obj)
    nConditionalDefiniensList: PList;
    nOtherwise: DefExpressionPtr;
    \textbf{constructor} \ \textit{Init}(\textbf{const} \ \textit{aPos: Position}; \ \textit{aLab: LabelPtr}; \ \textit{aPartialDefs: PList};
       aOtherwise: DefExpressionPtr);
    destructor Done; virtual;
  end;
```

§955 Mizar Parser DEFINITIONS 307

955. Constructor.

```
 \langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv \\ \textbf{constructor } \textit{ConditionalDefiniensObj.Init}(\textbf{const } aPos: \textit{Position}; \\ aLab: \textit{LabelPtr}; \textit{aPartialDefs: PList}; \textit{aOtherwise: DefExpressionPtr}); \\ \textbf{begin } \textit{inherited Init}(aPos, aLab, \textit{ConditionalDefiniens}); \textit{nConditionalDefiniensList} \leftarrow \textit{aPartialDefs}; \\ \textit{nOtherwise} \leftarrow \textit{aOtherwise}; \\ \textbf{end}; \\ \textbf{destructor } \textit{ConditionalDefiniensObj.Done}; \\ \textbf{begin if } \textit{nOtherwise} \neq \textbf{nil then } \textit{dispose}(\textit{nOtherwise}, \textit{Done}); \\ \textit{dispose}(\textit{nConditionalDefiniensList}, \textit{Done}); \textit{inherited Done}; \\ \textbf{end}; \\ \end{cases}
```

956. Mode definitions. Mizar was heavily inspired by ALGOL, and even borrows ALGOL's terminology for types ("modes"). These are "soft types", which are predicates in the ambient logic.

However, we need to establish the well-definedness of types (i.e., they are inhabited by at least one term), or else we end up in "free logic". For example, if EmptyType is a hypothetical empty type, then for x being EmptyType holds P[x] is always true, and ex x being EmptyType st P[x] is always false. The clever Mizar user can abuse this, and end up compromising the soundness of classical logic. To avert catastrophe, we require proving there exists at least one term of the newly defined type.

```
Mode-Definition = "mode" Mode-Pattern
  ([Specification]["means" Definiens]";" Correctness-Conditions | "is" Type-Expression ";")
  { Mode-Property } .
Mode-Pattern = Mode-Symbol [ "of" Loci ] .
Mode-Symbol = Symbol | "set" .
Mode-Synonym = "synonym" Mode-Pattern "for" Mode-Pattern ";" .
Mode-Property = "sethood" Justification ";" .
\langle \text{ Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  ModeDefinitionSort = (defExpandableMode, defStandardMode);
  ModeDefinitionPtr = \uparrow ModeDefinitionObj;
  ModeDefinitionObj = \mathbf{object} \ (MObject)
    nDefKind: ModeDefinitionSort;
    nDefModePos: Position;
    nDefModePattern: ModePatternPtr;
    nRedefinition: boolean:
    constructor Init(const aPos: Position; aDefKind: ModeDefinitionSort; aRedef: boolean;
       aPattern: ModePatternPtr);
    destructor Done; virtual;
  end:
      Constructor.
957.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor ModeDefinitionObj.Init(const aPos: Position; aDefKind: ModeDefinitionSort;
                                      aRedef: boolean; aPattern: ModePatternPtr);
  begin nDefKind \leftarrow aDefKind; nDefModePos \leftarrow aPos; nRedefinition \leftarrow aRedef;
  nDefModePattern \leftarrow aPattern;
  end;
destructor ModeDefinitionObj.Done;
  begin dispose(nDefModePattern, Done);
  end:
```

308 DEFINITIONS Mizar Parser $\S 958$

```
Expandable mode definitions. These are simple "abbreviations" of modes which are of the form
"mode \langle type \ name \rangle is \langle adjective_1 \rangle \cdots \langle adjective_n \rangle \langle type \rangle", i.e., just a stack of adjectives atop a type.
\langle \text{ Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  Expandable Mode Definition Ptr = \uparrow Expandable Mode Definition Obj;
  ExpandableModeDefinitionObj = \mathbf{object} \ (ModeDefinitionObj)
    nExpansion: TypePtr;
    constructor Init(const aPos: Position; aPattern: ModePatternPtr; aExp: TypePtr);
    destructor Done; virtual;
  end;
959. Constructor.
\langle Implementation for wsmarticle.pas 854\rangle +=
constructor ExpandableModeDefinitionObj.Init(const aPos: Position;
       aPattern: ModePatternPtr; aExp: TypePtr);
  begin inherited Init(aPos, defExpandableMode, false, aPattern); nExpansion <math>\leftarrow aExp;
destructor ExpandableModeDefinitionObj.Done;
  begin dispose (nExpansion, Done); inherited Done;
  end;
960.
       Standard mode definitions.
\langle \text{ Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  StandardModeDefinitionPtr = \uparrow StandardModeDefinitionObj;
  StandardModeDefinitionObj = \mathbf{object} \ (ModeDefinitionObj)
    nSpecification: TypePtr;
    nDefiniens: DefiniensPtr;
    constructor Init(const aPos: Position; aRedef: boolean; aPattern: ModePatternPtr;
       aSpec: TypePtr; aDef: DefiniensPtr);
    destructor Done; virtual;
  end;
961. Constructor.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor StandardModeDefinitionObj.Init(const aPos: Position;
       aRedef: boolean; aPattern: ModePatternPtr; aSpec: TypePtr; aDef: DefiniensPtr);
  begin inherited Init(aPos, defStandardMode, aRedef, aPattern); nSpecification <math>\leftarrow aSpec;
  nDefiniens \leftarrow aDef;
  end;
destructor StandardModeDefinitionObj.Done;
  begin dispose (nSpecification, Done); dispose (nDefiniens, Done); inherited Done;
  end:
```

§962 Mizar Parser DEFINITIONS 309

962. Attribute definitions. Attributes, like predicates, do not need to worry about correctness conditions. It's only when we want to use them like adjectives on a type that we need to worry, but that's a registration block concern.

```
Attribute-Definition = "attr" Attribute-Pattern "means" Definiens ";" Correctness-Conditions .
Attribute-Pattern = Locus "is" [ Attribute-Loci ] Attribute-Symbol .
\langle \text{ Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  AttributeDefinitionPtr = \uparrow AttributeDefinitionObj;
  AttributeDefinitionObj = \mathbf{object} (MObject)
    nDefAttrPos: Position;
    nDefAttrPattern\colon\thinspace AttributePatternPtr;
    nRedefinition: boolean;
    nDefiniens: DefiniensPtr;
    constructor Init(const aPos: Position; aRedef: boolean; aPattern: AttributePatternPtr;
       aDef: DefiniensPtr);
    destructor Done; virtual;
  end;
963. Constructor.
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
constructor AttributeDefinitionObj.Init(const aPos: Position;
       aRedef: boolean; aPattern: AttributePatternPtr; aDef: DefiniensPtr);
  begin nDefAttrPos \leftarrow aPos; nRedefinition \leftarrow aRedef; nDefAttrPattern \leftarrow aPattern;
  nDefiniens \leftarrow aDef;
  end;
destructor AttributeDefinitionObj.Done;
  begin dispose(nDefAttrPattern, Done); dispose(nDefiniens, Done);
  end;
964. Predicate definitions. Predicates are among the less demanding of the definitions: they are always
well-defined, so we do not need to worry about correctness conditions.
Predicate-Definition = "pred" Predicate-Pattern [ "means" Definiens ] ";"
Correctness-Conditions { Predicate-Property } .
Predicate-Pattern = [ Loci ] Predicate-Symbol [ Loci ] .
Predicate-Property = ("symmetry" | "asymmetry" | "connectedness" | "reflexivity" | "irreflexivity")
  Justification ";" .
Predicate-Synonym = "synonym" Predicate-Pattern "for" Predicate-Pattern ";" .
Predicate-Antonym = "antonym" Predicate-Pattern "for" Predicate-Pattern ";" .
Predicate-Symbol = Symbol | "=" .
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  PredicateDefinitionPtr = \uparrow PredicateDefinitionObj;
  PredicateDefinitionObj = \mathbf{object} (MObject)
    nDefPredPos: Position;
    nDefPredPattern: PredicatePatternPtr;
    nRedefinition: boolean;
    nDefiniens: DefiniensPtr;
    constructor Init(const aPos: Position; aRedef: boolean; aPattern: PredicatePatternPtr;
       aDef: DefiniensPtr);
    destructor Done; virtual;
  end;
```

310 DEFINITIONS Mizar Parser §965

965. Constructor.

```
⟨ Implementation for wsmarticle.pas 854⟩ +≡
constructor PredicateDefinitionObj.Init(const\ aPos:\ Position;
aRedef:\ boolean;\ aPattern:\ PredicatePatternPtr;\ aDef:\ DefiniensPtr);
begin nDefPredPos \leftarrow aPos;\ nRedefinition \leftarrow aRedef;\ nDefPredPattern \leftarrow aPattern;
nDefiniens \leftarrow aDef;
end;
destructor PredicateDefinitionObj.Done;
begin dispose(nDefPredPattern,Done);\ dispose(nDefiniens,Done);
end;
```

966. Functor definitions. We can also define new terms. Well, they introduce "term constructors" (constructors for terms). Mizar calls these guys "functors".

Functor definitions need to establish the well-definedness of the new term constructor. What this means depends on whether we define the new term using "means" or "equals", i.e.,

- (1) "\(\langle new term \rangle means \langle formula \rangle"\)" requires proving the existence and uniqueness of the new term;
- (2) "\(\lambda rew \text{ term}\rangle \text{ equals \(\lambda term \text{ expression}\rangle\)" requires proving the new term has the given type.

Why do we need to prove well-definedness? Well, classical logic requires proving there exists a model for a theory, so our hands are tied. If we removed this restriction, then we'd end up with something called "free logic", which is... weird.

```
Functor-Definition = "func" Functor-Pattern [ Specification ]
  [ ( "means" | "equals" ) Definiens ] ";"
  Correctness-Conditions { Functor-Property } .
Functor-Pattern = [ Functor-Loci ] Functor-Symbol [ Functor-Loci ]
  Left-Functor-Bracket Loci Right-Functor-Bracket .
Functor-Property = ( "commutativity" | "idempotence" | "involutiveness" | "projectivity" )
  Justification ";" .
Functor-Synonym = "synonym" Functor-Pattern "for" Functor-Pattern ";" .
Functor-Loci = Locus | "(" Loci ")" .
Functor-Symbol = Symbol .
Left-Functor-Bracket = Symbol | "{" | "[" .
Right-Functor-Bracket = Symbol | "}" | "]" .
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  FunctorDefinitionPtr = \uparrow FunctorDefinitionObj;
  FunctorDefinitionObj = \mathbf{object} (MObject)
    nDefFuncPos: Position;
    nDefFuncPattern: FunctorPatternPtr;
    nRedefinition: boolean;
    nSpecification: TypePtr;
    nDefiningWay: HowToDefine;
    nDefiniens: DefiniensPtr;
    constructor Init(const aPos: Position; aRedef: boolean; aPattern: FunctorPatternPtr;
      aSpec: TypePtr; aDefWay: HowToDefine; aDef: DefiniensPtr);
    destructor Done; virtual;
  end;
```

§967 Mizar Parser DEFINITIONS 311

```
967. Constructor.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor FunctorDefinitionObj.Init(const aPos: Position; aRedef: boolean;
      aPattern: FunctorPatternPtr; aSpec: TypePtr; aDefWay: HowToDefine; aDef: DefiniensPtr);
  begin nDefFuncPos \leftarrow aPos; nRedefinition \leftarrow aRedef; nDefFuncPattern \leftarrow aPattern;
  nSpecification \leftarrow aSpec; nDefiningWay \leftarrow aDefWay; nDefiniens \leftarrow aDef;
  end:
destructor FunctorDefinitionObj.Done;
  begin dispose(nDefFuncPattern, Done); dispose(nDefiniens, Done);
  end:
968. Notation block. We can recall the syntax for notation blocks.
Notation-Block = "notation" { Loci-Declaration | Notation-Declaration } "end" .
Notation-Declaration = Mode-Synonym
   Functor-Synonym
    Attribute-Synonym | Attribute-Antonym
   Predicate-Synonym | Predicate-Antonym .
Mode-Synonym = "synonym" Mode-Pattern "for" Mode-Pattern ";" .
Functor-Synonym = "synonym" Functor-Pattern "for" Functor-Pattern ";" .
Predicate-Synonym = "synonym" Predicate-Pattern "for" Predicate-Pattern ";" .
Predicate-Antonym = "antonym" Predicate-Pattern "for" Predicate-Pattern ";" .
Attribute-Synonym = "synonym" Attribute-Pattern "for" Attribute-Pattern ";" .
Attribute-Antonym = "antonym" Attribute-Pattern "for" Attribute-Pattern ";" .
The reader will observe all these notation items relate a new pattern which is either a synonym or antonym
for an old pattern. That is to say, we only need two patterns to store as data in a notation item node in the
abstract syntax tree.
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  NotationDeclarationPtr = \uparrow NotationDeclarationObj;
  NotationDeclarationObj = \mathbf{object} \ (mObject)
    nNotationPos: Position:
    nNotationSort: ItemKind;
    nOriginPattern, nNewPattern: PatternPtr;
    constructor Init(const aPos: Position; aNSort: ItemKind; aNewPatt, aOrigPatt: PatternPtr);
    destructor Done; virtual;
  end;
969. Constructor.
\langle Implementation for wsmarticle.pas 854\rangle + \equiv
constructor NotationDeclarationObj.Init(const aPos: Position; aNSort: ItemKind;
      aNewPatt, aOrigPatt: PatternPtr);
  begin nNotationPos \leftarrow aPos; nNotationSort \leftarrow aNSort; nOriginPattern \leftarrow aOrigPatt;
  nNewPattern \leftarrow aNewPatt;
destructor NotationDeclarationObj.Done;
  begin dispose(nOriginPattern, Done); dispose(nNewPattern, Done);
  end;
```

312 DEFINITIONS Mizar Parser §970

```
970.
      Assumptions in a definition block. The syntax for assumptions in a definition block looks like:
Assumption = Single-Assumption | Collective-Assumption | Existential-Assumption .
Single-Assumption = "assume" Proposition ";" .
Collective-Assumption = "assume" Conditions ";" .
Existential-Assumption = "given" Qualified-Variables [ "such" Conditions ] ";" .
Conditions = "that" Proposition { "and" Proposition } .
Proposition = [ Label-Identifier ":" ] Sentence .
Sentence = Formula-Expression .
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  AssumptionKind = (Single Assumption, Collective Assumption, Existential Assumption);
  AssumptionPtr = \uparrow AssumptionObj;
  AssumptionObj = \mathbf{object} \ (MObject)
    nAssumptionPos: Position;
    nAssumptionSort: AssumptionKind;
    constructor Init(const aPos: Position; aSort: AssumptionKind);
  end;
971. Constructor.
\langle Implementation for wsmarticle.pas 854\rangle +=
\textbf{constructor} \ \textit{AssumptionObj.Init} (\textbf{const} \ \textit{aPos:} \ \textit{Position}; \ \textit{aSort:} \ \textit{AssumptionKind});
  begin nAssumptionPos \leftarrow aPos; nAssumptionSort \leftarrow aSort;
  end:
972. Single assumption. When a definition has a single assumption, i.e., a single (usually labeled)
formula.
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  Single Assumption Ptr = \uparrow Single Assumption Obj;
  Single Assumption Obj = \mathbf{object} (Assumption Obj)
    nProp: PropositionPtr;
    constructor Init (const aPos: Position; aProp: PropositionPtr);
    destructor Done; virtual;
  end;
973.
      Constructor.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor SingleAssumptionObj.Init(const aPos: Position; aProp: PropositionPtr);
  begin inherited Init(aPos, Single Assumption); nProp \leftarrow aProp;
  end:
destructor SingleAssumptionObj.Done;
  begin dispose(nProp, Done);
  end:
```

§974 Mizar Parser DEFINITIONS 313

```
974.
       Collective assumption. This describes the case when the assumption is "assume C_1 and ... and
C_n".
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  CollectiveAssumptionPtr = \uparrow CollectiveAssumptionObj;
  CollectiveAssumptionObj = \mathbf{object} \ (AssumptionObj)
    nConditions: PList;
    constructor Init(const aPos: Position; aProps: PList);
    destructor Done; virtual;
  end;
975. Constructor.
\langle Implementation for wsmarticle.pas 854\rangle +=
constructor CollectiveAssumptionObj.Init(const aPos: Position; aProps: PList);
  begin inherited Init(aPos, Collective Assumption); nConditions <math>\leftarrow aProps;
  end;
destructor CollectiveAssumptionObj.Done;
  begin dispose(nConditions, Done);
  end;
976.
      Existential assumption. I must confess I am surprised to see an existential assumption node being
a subclass of a collective assumption node.
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  ExistentialAssumptionPtr = \uparrow ExistentialAssumptionObj;
  ExistentialAssumptionObj = \mathbf{object} (CollectiveAssumptionObj)
    nQVars: PList;
    constructor Init(const aPos: Position; aQVars, aProps: PList);
    destructor Done; virtual;
  end;
977. Constructor.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor Existential Assumption Obj. Init(const aPos: Position; aQVars, aProps: PList);
  begin AssumptionObj.Init(aPos, CollectiveAssumption); nConditions \leftarrow aProps; nQVars \leftarrow aQVars;
  end;
destructor Existential Assumption Obj. Done;
  begin dispose(nQVars, Done); inherited Done;
  end;
```

314 DEFINITIONS Mizar Parser $\S978$

```
Correctness conditions. The syntax for correctness conditions:
Correctness-Conditions = {Correctness-Condition} [ "correctness" Justification ";" ] .
Correctness-Condition =
  ( "existence" | "uniqueness" | "coherence" | "compatibility" | "consistency" | "reducibility" )
  Justification ";" .
We begin with an abstract base class for correctness conditions.
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  CorrectnessPtr = \uparrow CorrectnessObj;
  CorrectnessObj = \mathbf{object} \ (MObject)
    nCorrCondPos: Position;
    nJustification: JustificationPtr;
    constructor Init(const aPos: Position; aJustification: JustificationPtr);
    destructor Done; virtual;
  end;
979. Constructor.
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
constructor CorrectnessObj.Init(const aPos: Position; aJustification: JustificationPtr);
  begin nCorrCondPos \leftarrow aPos; nJustification \leftarrow aJustification;
  end;
destructor CorrectnessObj.Done;
  begin dispose(nJustification, Done);
  end:
980. Correctness condition. For the correctness condition associated with a definition or registration,
we have this Correctness Condition object. When we need multiple correctness conditions, we extend it with
a subclass.
\langle \text{ Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  CorrectnessConditionPtr = \uparrow CorrectnessConditionObj;
  CorrectnessConditionObj = \mathbf{object} (CorrectnessObj)
    nCorrCondSort: CorrectnessKind;
    constructor Init(const aPos: Position; aSort: CorrectnessKind; aJustification: JustificationPtr);
    destructor Done; virtual;
  end;
981. Constructor.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor CorrectnessConditionObj.Init(const aPos: Position;
       aSort: CorrectnessKind; aJustification: JustificationPtr);
  begin inherited Init(aPos, aJustification); nCorrCondSort \leftarrow aSort;
destructor CorrectnessConditionObj.Done;
  begin inherited Done;
  end;
```

 $\S982$ Mizar Parser DEFINITIONS 315

Multiple correctness conditions. For, e.g., functors which require proving both "existence" and "uniqueness", we have a Correctness Conditions class. This extends the [singular] Correctness Condition $\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv$ $CorrectnessConditionsSet = \mathbf{set} \ \mathbf{of} \ CorrectnessKind;$ $CorrectnessConditionsPtr = \uparrow CorrectnessConditionsObj;$ $CorrectnessConditionsObj = \mathbf{object} \ (CorrectnessObj)$ nConditions: Correctness Conditions Set;**constructor** *Init*(**const** *aPos*: *Position*; **const** *aConds*: *CorrectnessConditionsSet*; aJustification: JustificationPtr): destructor Done; virtual; end; 983. Constructor. \langle Implementation for wsmarticle.pas $854 \rangle + \equiv$ **constructor** CorrectnessConditionsObj.Init(**const** aPos: Position; **const** aConds: CorrectnessConditionsSet; a Justification: Justification Ptr);**begin** inherited $Init(aPos, aJustification); nConditions \leftarrow aConds;$ **destructor** CorrectnessConditionsObj.Done; **begin** inherited Done; end: 984. **Definition properties.** The grammar for properties in a definition looks like: Mode-Property = "sethood" Justification ";" . Functor-Property = ("commutativity" | "idempotence" | "involutiveness" | "projectivity") Justification ";" . Predicate-Property = ("symmetry" | "asymmetry" | "connectedness" | "reflexivity" | "irreflexivity") Justification ";" . We see these are all, more or less, "the same": we have a "kind" of property and a justification. We recall (§720) that we have already introduced the "kind" of properties. So the class describing a definition property node in the abstract syntax tree is: $\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv$ $PropertyPtr = \uparrow PropertyObj;$

constructor Init(**const** aPos: Position; aSort: PropertyKind; aJustification: JustificationPtr);

PropertyObj = **object** (MObject) nPropertyPos: Position; nPropertySort: PropertyKind; nJustification: JustificationPtr;

destructor Done; virtual;

end;

316 DEFINITIONS Mizar Parser §985

985. Constructor.

```
\langle Implementation for wsmarticle.pas 854\rangle +\equiv constructor PropertyObj.Init (const aPos: Position; aSort: PropertyKind; aJustification: JustificationPtr); begin nPropertyPos \leftarrow aPos; nPropertySort \leftarrow aSort; nJustification \leftarrow aJustification; end; destructor PropertyObj.Done; begin inheritedDone; end;
```

§986 Mizar Parser REGISTRATIONS 317

Section 21.9. REGISTRATIONS

end:

986. There are three "main" types of registrations, which are "cluster registrations" (because they all involve the "cluster" keyword):

- (1) Existential registrations are of the form "cluster $\langle attribute \rangle$ for $\langle type \rangle$ " and establishes that a given attribute can act as an adjective for the type.
- (2) Conditional registrations are of the form "cluster $\langle attribute_1 \rangle$ -> $\langle attribute_2 \rangle$ for $\langle type \rangle$ " which tells Mizar that when $\langle attribute_1 \rangle$ is established for a term, then Mizar can automatically add $\langle attribute_2 \rangle$ for the term
- (3) Functorial registrations are of the form "cluster $\langle term \rangle \rightarrow \langle attribute \rangle$ [for $\langle type \rangle$]" which will automatically add an attribute to a term.

We also have three lesser registrations which are still important:

- (1) Sethood registrations, establishes a type can be used as a set in a Fraenkel term.
- (2) Reduction registration, which allows Mizar's term rewriting module to use this rule when reasoning about things.
- (3) Identification registration, which allows Mizar to identify terms of different types.

```
Cluster-Registration = Existential-Registration
   Conditional-Registration
   Functorial-Registration .
Existential-Registration = "cluster" Adjective-Cluster "for" Type-Expression ";"
  Correctness-Conditions .
Adjective-Cluster = { Adjective } .
Adjective = [ "non" ] [ Adjective-Arguments ] Attribute-Symbol .
Conditional-Registration = "cluster" Adjective-Cluster "->" Adjective-Cluster "for" Type-Expression ";"
  Correctness-Conditions
Functorial-Registration = "cluster" Term-Expression "->" Adjective-Cluster [ "for" Type-Expression ] ";"
  Correctness-Conditions .
Identify-Registration = "identify" Functor-Pattern "with" Functor-Pattern
    [ "when" Locus "=" Locus { "," Locus "=" Locus } ] ";"
  Correctness-Conditions .
Property-Registration = "sethood" "of" Type-Expression Justification ";" .
Reduction-Registration = "reduce" Term-Expression "to" Term-Expression ";"
  Correctness-Conditions .
987. Cluster registration. We have a base class for the three types of cluster registrations.
\langle \text{ Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  Cluster Registration Kind = (Existential Registration, Conditional Registration, Functorial Registration);
  ClusterPtr = \uparrow ClusterObj;
  ClusterObj = \mathbf{object} \ (MObject)
    nClusterPos: Position;
    nClusterKind: ClusterRegistrationKind;
    nConsequent: PList;
    nClusterType: TypePtr;
    constructor Init(const aPos: Position; aKind: ClusterRegistrationKind; aCons: PList;
       aTyp: TypePtr);
    destructor Done; virtual;
```

318 REGISTRATIONS Mizar Parser §988

```
988.
       Constructor.
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
constructor ClusterObj.Init(const aPos: Position;
       aKind: ClusterRegistrationKind; aCons: PList; aTyp: TypePtr);
  begin nClusterPos \leftarrow aPos; nClusterKind \leftarrow aKind; nConsequent \leftarrow aCons; nClusterType \leftarrow aTyp;
destructor ClusterObj.Done;
  begin dispose(nConsequent, Done);
  end;
989.
       Existential cluster. We register the fact there always exists a term of a given type satisfying an
attribute (e.g., "empty" for "set" means there always exists an empty set; registering the existential cluster
"non empty" for "set" means there always exists a nonempty set). This means the attribute may henceforth
be used as an adjective on the type.
\langle \text{ Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  EClusterPtr = \uparrow EClusterObj;
  EClusterObj = \mathbf{object} \ (ClusterObj)
    constructor Init(const aPos: Position; aCons: PList; aTyp: TypePtr);
    destructor Done; virtual;
  end;
      Constructor. There are no additional fields to an existential cluster object, so it literally passes the
parameters onto the superclass's constructor.
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
constructor EClusterObj.Init(const aPos: Position; aCons: PList; aTyp: TypePtr);
  begin ClusterObj.Init(aPos, ExistentialRegistration, aCons, aTyp);
  end;
destructor EClusterObj.Done;
  begin if nClusterType \neq nil then dispose(nClusterType, Done);
  inherited Done:
  end;
       Conditional cluster. For example "empty sets" are always "finite sets". This requires tracking the
antecedent ("empty"), and the superclass tracks the consequents ("finite").
\langle \text{ Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  CClusterPtr = \uparrow CClusterObj;
  CClusterObj = \mathbf{object} (ClusterObj)
    nAntecedent: PList;
    constructor Init(const aPos: Position; aAntec, aCons: PList; aTyp: TypePtr);
    destructor Done; virtual;
  end;
       Constructor.
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
constructor CClusterObj.Init(const aPos: Position; aAntec, aCons: PList; aTyp: TypePtr);
  begin ClusterObj.Init(aPos, ConditionalRegistration, aCons, aTyp); nAntecedent <math>\leftarrow aAntec;
  end;
destructor CClusterObj.Done;
  begin dispose (nAntecedent, Done); inherited Done;
```

end;

§993 Mizar Parser REGISTRATIONS 319

Functorial cluster. The generic form a functorial registrations associated to a term some cluster of adjectives. We need to track the term, but the superclass can manage the cluster of adjectives. $\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv$ $FClusterPtr = \uparrow FClusterObj;$ $FClusterObj = \mathbf{object} (ClusterObj)$ nClusterTerm: TermPtr;constructor Init(const aPos: Position; aTrm: TermPtr; aCons: PList; aTyp: TypePtr); **destructor** Done; virtual; end; 994. Constructor. \langle Implementation for wsmarticle.pas 854 $\rangle +=$ **constructor** FClusterObj.Init(**const** aPos: Position; aTrm: TermPtr; aCons: PList; aTyp: TypePtr); **begin** ClusterObj.Init(aPos, FunctorialRegistration, aCons, aTyp); nClusterTerm \leftarrow aTrm; end: **destructor** FClusterObj.Done; **begin if** $nClusterTerm \neq nil$ **then** Dispose(nClusterTerm, Done);if $nClusterType \neq nil then dispose(nClusterType, Done);$ inherited Done; end: **Loci equality.** This is used in identification registrations. $\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv$ $LociEqualityPtr = \uparrow LociEqualityObj;$ $LociEqualityObj = \mathbf{object} \ (mObject)$ nEqPos: Position;nLeftLocus, nRightLocus: LocusPtr;**constructor** Init(**const** aPos: Position; aLeftLocus, aRightLocus: LocusPtr); **destructor** Done; virtual; end; 996. Constructor. \langle Implementation for wsmarticle.pas $854 \rangle + \equiv$ **constructor** LociEqualityObj.Init(**const** aPos: Position; aLeftLocus, aRightLocus: LocusPtr); **begin** $nEqPos \leftarrow aPos$; $nLeftLocus \leftarrow aLeftLocus$; $nRightLocus \leftarrow aRightLocus$; end; **destructor** LociEqualityObj.Done; **begin** Dispose(nLeftLocus, Done); dispose(nRightLocus, Done);end:

320 REGISTRATIONS Mizar Parser $\S997$

```
Identification registration. Term identification was first introduced in Artur Korniłowicz's "How
to define terms in Mizar effectively" (in A. Grabowski and A. Naumowicz (eds.), Computer Reconstruction
of the Body of Mathematics, issue of Studies in Logic, Grammar and Rhetoric 18 no.31 (2009), pp. 67-
77). See also §2.7 of Adam Grabowski, Artur Korniłowicz, and Adam Naumowicz's "Mizar in a Nutshell"
(doi:10.6092/issn.1972-5787/1980) for user-oriented details.
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  IdentifyRegistrationPtr = \uparrow IdentifyRegistrationObj;
  IdentifyRegistrationObj = \mathbf{object} \ (mObject)
    nIdentifyPos: Position;
    nOriginPattern, nNewPattern: PatternPtr;
    nEqLociList: PList;
    constructor Init(const aPos: Position; aNewPatt, aOriqPatt: PatternPtr; aEqList: PList);
    destructor Done; virtual;
  end;
998.
       Constructor.
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
\textbf{constructor} \ \textit{IdentifyRegistrationObj.Init} (\textbf{const} \ \textit{aPos: Position};
       aNewPatt, aOriqPatt: PatternPtr; aEqList: PList);
  begin nIdentifyPos \leftarrow aPos; nOriginPattern \leftarrow aOrigPatt; nNewPattern \leftarrow aNewPatt;
  nEqLociList \leftarrow aEqList:
  end;
destructor IdentifyRegistrationObj.Done;
  begin dispose(nOriginPattern, Done); dispose(nNewPattern, Done);
  if nEqLociList \neq nil then dispose(nEqLociList, Done);
  end;
       Property registration. These were introduced in Mizar to facilitated registering "sethood" for
types. Thus far, only the "sethood" property is handled in this registration.
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  PropertyRegistrationPtr = \uparrow PropertyRegistrationObj;
  PropertyRegistrationObj = \mathbf{object} \ (mObject)
    nPropertyPos: Position;
    nPropertySort: PropertyKind;
    constructor Init (const aPos: Position; aKind: PropertyKind);
    destructor Done; virtual;
  end;
1000. Constructor.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor PropertyRegistrationObj.Init(const aPos: Position; aKind: PropertyKind);
  begin nPropertyPos \leftarrow aPos; nPropertySort \leftarrow aKind;
{\bf destructor}\ \textit{PropertyRegistrationObj.Done};
  begin end;
```

 $\{1001$ Mizar Parser REGISTRATIONS 321

```
Sethood registration. Artur Korniłowicz's "Sethood Property in Mizar" (in Joint Proc. FMM
and LML Workshops, 2019, ceur-ws.org/Vol-2634/FMM3.pdf) introduces this "sethood" property. It's the
first (and, so far, only) property registration in Mizar.
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  SethoodRegistrationPtr = \uparrow SethoodRegistrationObj;
  SethoodRegistrationObj = \mathbf{object} (PropertyRegistrationObj)
    nSethoodType: TypePtr;
    nJustification: JustificationPtr;
    constructor Init(const aPos: Position; aKind: PropertyKind; aType: TypePtr);
    destructor Done: virtual:
  end;
1002. Constructor.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor SethoodRegistrationObj.Init(const aPos: Position;
       aKind: PropertyKind; aType: TypePtr);
  begin inherited Init(aPos, aKind); nSethoodType \leftarrow aType; nJustification \leftarrow nil;
  end;
destructor SethoodRegistrationObj.Done;
  begin dispose(nSethoodType, Done); dispose(nJustification, Done); inherited Done;
  end:
        Reduce registration. These were introduced, I think, in Artur Korniłowicz's "On rewriting rules
in Mizar" (J. Autom. Reason. 50 no.2 (2013) 203-210, doi:10.1007/s10817-012-9261-6). These extend
the checker with new term rewriting rules.
\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  ReduceRegistrationPtr = \uparrow ReduceRegistrationObj;
  ReduceRegistrationObj = \mathbf{object} (MObject)
    nReducePos: Position;
    nOriginTerm, nNewTerm: TermPtr;
    constructor Init(const aPos: Position; aOrigTerm, aNewTerm: TermPtr);
    destructor Done; virtual;
  end;
1004.
        Constructor.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor ReduceRegistrationObj.Init(const aPos: Position; aOrigTerm, aNewTerm: TermPtr);
  begin nReducePos \leftarrow aPos; nOriginTerm \leftarrow aOrigTerm; nNewTerm \leftarrow aNewTerm;
  end:
destructor ReduceRegistrationObj.Done;
  begin dispose(nOriginTerm, Done); dispose(nNewTerm, Done);
  end;
```

322 HELPER FUNCTIONS Mizar Parser §1005

Section 21.10. HELPER FUNCTIONS

```
1005. Capitlization checks if the first character c is lowercase. If so, then set the leading character to be
c \leftarrow c - (ord(\hat{a}) - ord(\hat{a})). But it leaves the rest of the string untouched.
\langle Implementation for wsmarticle.pas 854\rangle + \equiv
function CapitalizeName (aName: string): string;
  begin result \leftarrow aName;
  \mathbf{if} \ \ aName[1] \in [\texttt{`a`..`z'}] \ \mathbf{then} \ \ dec(Result[1], ord(\texttt{`a'}) - ord(\texttt{`A'}))
  end;
1006. Uncapitalizing works in the opposite direction, setting the first letter c of a string to be c \leftarrow
c + (ord(\hat{a}) - ord(\hat{A})). Observe capitalizing and uncapitalizing are "nearly inverses" of each other:
CapitalizeName(UncapitalizeName(CapitalizeName(s))) = CapitalizeName(s), and similarly we find
UncapitalizeName(CapitalizeName(UncapitalizeName(s))) = UncapitalizeName(s).
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function UncapitalizeName(aName: string): string;
  begin result \leftarrow aName;
  if aName[1] \in [`A`..`Z`] then inc(Result[1], ord(`a`) - ord(`A`))
1007.
         We will be populating global variables tracking names of identifiers, modes, and other syntactic
classes.
\langle Global variables publicly declared in wsmarticle.pas 1007\rangle \equiv
\mathbf{var}\ IdentifierName, AttributeName, StructureName, ModeName, PredicateName, FunctorName,
         SelectorName, LeftBracketName, RightBracketName, MMLIdentifierName: array of string;
This code is used in section 850.
1008. We will want to initialize these global variables based on previous passes of the scanner.
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
procedure InitScannerNames;
  var i, lCnt, lNr: integer; lDct: text; lInFile: XMLInStreamPtr; lKind, lDummy: AnsiChar;
    lString: string;
  begin (Populate global variables with XML entities 1009);
  ⟨Reset reserved keywords 1011⟩;
    { Identifiers }
  ⟨Initialize identifier names from .idx file 1012⟩;
  end;
```

 $\{1009$ Mizar Parser HELPER FUNCTIONS 323

1009. We need to initialize the length for each of these arrays. Even a crude approximation works, like the total number of lines in the .dct file. Then we transform each line of the *lDct* (dictionary file) into appropriate entries of the relevant array.

```
⟨ Populate global variables with XML entities 1009⟩ ≡
    assign(lDct, MizFileName + ´.dct´); reset(lDct); lCnt ← 0;
while ¬seekEof(lDct) do
    begin readln(lDct); inc(lCnt);
    end;
setlength(AttributeName, lCnt); setlength(StructureName, lCnt); setlength(ModeName, lCnt);
setlength(PredicateName, lCnt); setlength(FunctorName, lCnt); setlength(SelectorName, lCnt);
setlength(LeftBracketName, lCnt); setlength(RightBracketName, lCnt);
setlength(MMLIdentifierName, lCnt); reset(lDct);
while ¬seekEof(lDct) do
    begin readln(lDct, lKind, lNr, lDummy, lString); ⟨Store XML version of vocabulary word 1010⟩;
end;
close(lDct)
This code is used in section 1008.
```

1010. We have read in from the ".dct" file one line. The first 148 lines of a ".dct" file consists of the reserved keywords for Mizar. A random example of the last few lines of such a file look like:

```
A36 VECTSP_4
A37 ORDINAL1
A38 CARD_FIL
A39 RANKNULL
A40 VECTSP_1
A41 VECTSP_6
A42 VECTSP13
A43 ALGSTR_0
A44 HALLMAR1
A45 MATROIDO
```

This code is used in section 1009.

So we read the first leading letter of a line into lKind, then the number into lNr, the space is stuffed into lDummy, and the remainder of the line is placed in lString.

```
 \langle \text{Store XML version of vocabulary word } 1010 \rangle \equiv \\ \text{case } lKind \text{ of} \\ \quad \land \land : MMLIdentifierName[lNr] \leftarrow QuoteStrForXML(lString); \\ \quad \land G : StructureName[lNr] \leftarrow QuoteStrForXML(lString); \\ \quad \land \land : ModeName[lNr] \leftarrow QuoteStrForXML(lString); \\ \quad \land \land : LeftBracketName[lNr] \leftarrow QuoteStrForXML(lString); \\ \quad \land \land : LeftBracketName[lNr] \leftarrow QuoteStrForXML(lString); \\ \quad \land \land : RightBracketName[lNr] \leftarrow QuoteStrForXML(lString); \\ \quad \land \land : PredicateName[lNr] \leftarrow QuoteStrForXML(lString); \\ \quad \land \land : SelectorName[lNr] \leftarrow QuoteStrForXML(lString); \\ \quad \land \land \land : AttributeName[lNr] \leftarrow QuoteStrForXML(lString); \\ \quad \text{endcases} \\ \end{aligned}
```

324 HELPER FUNCTIONS Mizar Parser $\S 1011$

1011. Preserve reserved keywords. We want to prevent the user from "overwriting" or "shadowing" the builtin primitive reserved words. This should probably be documented in the user-manual somewhere. The reserved words are: "strict", "set", "=", and the brackets [], braces {}, and parentheses (). Curiously, "object" is not considered a 'primitive' worth preserving.

```
\langle \operatorname{Reset \ reserved \ keywords \ 1011} \rangle \equiv \\ AttributeName[StrictSym] \leftarrow `\operatorname{strict'}; \ ModeName[SetSym] \leftarrow `\operatorname{set'}; \\ PredicateName[EqualitySym] \leftarrow `=`; \ LeftBracketName[SquareBracket] \leftarrow `[`; \\ LeftBracketName[CurlyBracket] \leftarrow ``{`; \ LeftBracketName[RoundedBracket] \leftarrow `(`; \\ RightBracketName[SquareBracket] \leftarrow `]`; \ RightBracketName[CurlyBracket] \leftarrow ``{}`; \\ RightBracketName[RoundedBracket] \leftarrow ``)` \\ \text{This code is used in section 1008}.
```

1012. The .idx file provides numbers for the local labels and article names referenced in an article.

```
⟨ Initialize identifier names from .idx file 1012⟩ ≡ assign(lDct, MizFileName + `.idx`); reset(lDct); lCnt \leftarrow 0;
while \neg seekEof(lDct) do
begin readln(lDct); inc(lCnt);
end;
close(lDct);
setlength(IdentifierName, lCnt); IdentifierName[0] \leftarrow ``;
lInFile \leftarrow new(XMLInStreamPtr, OpenFile(MizFileName + `.idx`)); lInFile↑.NextElementState;
lInFile↑.NextElementState;
while (lInFile.nState = eStart) \wedge (lInFile.nElName = XMLElemName[elSymbol]) do
begin lNr \leftarrow lInFile↑.GetIntAttr(`nr`); lString \leftarrow lInFile↑.GetAttr(`name`);
IdentifierName[lNr] \leftarrow lString; lInFile↑.NextElementState; lInFile↑.NextElementState;
end;
dispose(lInFile, Done)
This code is used in section 1008.
```

1013. We will want to obtain the name for an article ID number, provided it is a legal number (i.e., less than the dictionary for article ID numbers). This function looks up its entry in the *IdentifierName* array.

 $\{1014$ Mizar Parser WRITING WSM XML FILES 325

Section 21.11. WRITING WSM XML FILES

```
1014.
```

```
\langle \text{ Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  OutWSMizFilePtr = \uparrow OutWSMizFileObj;
  OutWSMizFileObj = \mathbf{object} (XMLOutStreamObj)
    nDisplayInformationOnScreen: boolean;
    nMizarAppearance: boolean;
    constructor OpenFile(const aFileName: string);
    constructor OpenFileWithXSL(const aFileName: string);
    destructor Done; virtual;
    procedure Out_TextProper(aWSTextProper: WSTextProperPtr); virtual;
    procedure Out_Block(aWSBlock : WSBlockPtr); virtual;
    procedure Out_Item(aWSItem: WSItemPtr); virtual;
    procedure Out_ItemContentsAttr(aWSItem: WSItemPtr); virtual;
    procedure Out_ItemContents(aWSItem: WSItemPtr); virtual;
    procedure Out_Variable(aVar : VariablePtr); virtual;
    procedure Out_Reserved Variable (a Var : VariablePtr); virtual;
    procedure Out_TermList(aTrmList: PList); virtual;
    procedure Out_Adjective(aAttr: AdjectiveExpressionPtr); virtual;
    procedure Out_AdjectiveList(aCluster: PList): virtual:
    procedure Out_Type(aTyp: TypePtr); virtual;
    procedure Out_ImplicitlyQualifiedVariable(aSeqm: ImplicitlyQualifiedSeqmentPtr); virtual;
    procedure Out_VariableSegment(aSegm : QualifiedSegmentPtr); virtual;
    procedure Out_PrivatePredicativeFormula(aFrm: PrivatePredicativeFormulaPtr); virtual;
    procedure Out_Formula(aFrm : FormulaPtr); virtual;
    procedure Out_Term(aTrm : TermPtr); virtual;
    procedure Out_SimpleTerm(aTrm : SimpleTermPtr); virtual;
    procedure Out_PrivateFunctorTerm(aTrm:PrivateFunctorTermPtr); virtual;
    procedure Out_InternalSelectorTerm(aTrm: InternalSelectorTermPtr); virtual;
    procedure Out_TypeList(aTypeList : PList); virtual;
    procedure Out_Locus(aLocus: LocusPtr); virtual;
    procedure Out_Loci(aLoci : PList); virtual;
    procedure Out_Pattern(aPattern: PatternPtr); virtual;
    procedure Out_Label(aLab : LabelPtr); virtual;
    procedure Out_Definiens(aDef : DefiniensPtr); virtual;
    procedure Out_ReservationSegment(aRes : ReservationSegmentPtr); virtual;
    procedure Out_SchemeNameInSchemeHead(aSch: SchemePtr); virtual;
    procedure Out_CompactStatement(aCStm : CompactStatementPtr; aBlock : wsBlockPtr); virtual;
    procedure Out_RegularStatement(aRStm: RegularStatementPtr; aBlock: wsBlockPtr); virtual;
    procedure Out_Proposition(aProp : PropositionPtr); virtual;
    procedure Out_LocalReference(aRef : LocalReferencePtr); virtual;
    procedure Out_References(aRefs: PList); virtual;
    procedure Out_Link(aInf : JustificationPtr); virtual;
    procedure Out_SchemeJustification(aInf: SchemeJustificationPtr): virtual;
    procedure Out_Justification(aInf: JustificationPtr; aBlock: wsBlockPtr); virtual;
  end;
```

326 WRITING WSM XML FILES Mizar Parser $\S1015$

1015. Constructor. The constructor OutWSMizFileObj.OpenFileWithXSL is not used anywhere, nor is the associated "wsmiz.xml" file present anywhere.

Importantly, the nMizarAppearance field controls whether the XML generated includes the raw lexeme string as an attribute in the XML elements or not.

The constructor OpenFileWithXSL is never used. The XML stylesheet wsmiz.xml does not seem to be present in the Mizar distribution.

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor OutWSMizFileObj.OpenFile(const aFileName: string);
  begin inherited\ OpenFile\ (aFileName);\ nMizarAppearance \leftarrow false;
  nDisplayInformationOnScreen \leftarrow false;
  end:
constructor OutWSMizFileObj.OpenFileWithXSL(const aFileName: string);
  begin inherited OpenFile(aFileName);
  OutString(`<?xml-stylesheet_type="text/xml"_href="file://`+MizFiles+`wsmiz.xml"?>`+#10);
  nMizarAppearance \leftarrow false;
  end:
destructor OutWSMizFileObj.Done;
  begin inherited Done;
  end;
        We can write the XML for a wsTextProper object (§856). This writes out the start tag, the children,
and the end-tag for the "text proper" and its contents. The RNG compact schema for this looks like:
TextProper = element Text-Proper {
  attribute idnr { xsd:integer },
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Item*
}
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_TextProper(aWSTextProper: WSTextProperPtr);
  var i: integer;
  begin with aWSTextProper \uparrow do
            { Write the start-tag }
    begin
    Out\_XElStart(BlockName[blMain]); Out\_XAttr(XMLAttrName[atArticleId], nArticleId);
    Out\_XAttr(XMLAttrName[atArticleExt], nArticleExt); Out\_PosAsAttrs(nBlockPos); Out\_XAttrEnd;
    for i \leftarrow 0 to nItems \uparrow. Count - 1 do Out\_Item(nItems.Items \uparrow [i]); \{ ...then write the children \}
    Out\_XElEnd(BlockName[blMain]);
    end;
  end;
```

 $\S1017$ Mizar Parser WRITING WSM XML FILES 327

1017. Writing a block out as XML works similarly: write the start-tag, then its children elements, then the end-tag.

```
Block = element Block {
  attribute kind { "Text-Proper" | "Now-Reasoning"
      "Hereby-Reasoning" | "Definitional-Block"
      "Notation-Block" | "Registration-Block" | "Case"
      "Suppose" | "Scheme-Block" },
  attribute idnr { xsd:integer },
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Item*
}
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_Block(aWSBlock: WSBlockPtr);
  var i: integer;
  begin with aWSBlock \uparrow do
    begin { write the start-tag }
    Out_XElStart(XMLElemName[elBlock]);
    Out\_XAttr(XMLAttrName[atKind], BlockName[nBlockKind]); CurPos \leftarrow nBlockPos;
    Out\_PosAsAttrs(nBlockPos); Out\_XIntAttr(XMLAttrName[atPosLine], nBlockEndPos.Line);
    Out\_XIntAttr(XMLAttrName[atPosCol], nBlockEndPos.Col); Out\_XAttrEnd;
    for i \leftarrow 0 to nItems \uparrow. Count - 1 do
      begin Out\_Item(nItems\uparrow.Items\uparrow[i]); end; { Then write the children }
    Out_XElEnd(XMLElemName[elBlock]);
    end;
  end;
```

1018. Writing a term list to XML amounts to just writing the terms as XML elements. They will be contained in a parent element, so there will be no ambiguity in their role.

```
Term-List = ( Term* ) 

\langle Implementation for wsmarticle.pas 854\rangle += 

procedure OutWSMizFileObj.Out\_TermList(aTrmList:PList);

var i: integer;

begin for i \leftarrow 0 to aTrmList\uparrow.Count - 1 do Out\_Term(aTrmList\uparrow.Items\uparrow[i]);

end;
```

328 WRITING WSM XML FILES Mizar Parser $\S1019$

1019. The XML for an adjective boils down to two cases:

Case 1 (negated attribute). Write a <NegatedAdjective> tag around the XML produced from case 2 for the positive version of the attribute.

Case 2 (positive attribute). Write the adjective, and its children are the [term] arguments to the adjective (if any — if there are none, then an empty-element will be produced).

```
PositiveAdjective = element Adjective {
  attribute nr { xsd:integer },
  attribute name { text },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer }
  Term*
}
Adjective = PositiveAdjective | element NegatedAdjective {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  PositiveAdjective
}
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_Adjective(aAttr : AdjectiveExpressionPtr);
  begin case aAttr\uparrow.nAdjectiveSort of
  wsAdjective: begin Out_XElStart(XMLElemName[elAdjective]);
    with AdjectivePtr(aAttr)\uparrow do
      begin Out\_XIntAttr(XMLAttrName[atNr], nAdjectiveSymbol);
      if nMizarAppearance then
        Out\_XAttr(XMLAttrName[atSpellinq], AttributeName[nAdjectiveSymbol]);
      Out\_PosAsAttrs(nAdjectivePos);
      if nArgs\uparrow.Count = 0 then Out\_XElEnd0
      else begin Out_XAttrEnd; Out_TermList(nArgs); Out_XElEnd(XMLElemName[elAdjective]);
      end;
    end;
  wsNegatedAdjective: begin Out_XElStart(XMLElemName[elNegatedAdjective]);
    with NegatedAdjectivePtr(aAttr)\uparrow do
      begin Out_PosAsAttrs(nAdjectivePos); Out_XAttrEnd; Out_Adjective(nArg);
    Out_XElEnd(XMLElemName[elNegatedAdjective]);
    end:
  endcases;
  end;
```

 $\S1020$ Mizar Parser WRITING WSM XML FILES 329

1020. Writing an adjective list to XML amounts to stuffing all the adjectives into an element. If there are no adjectives, it is the empty-element.

```
Adjective-Cluster = element Adjective-Cluster {
   attribute count { xsd:integer },
   Adjective*
}

⟨Implementation for wsmarticle.pas 854⟩ +≡
procedure OutWSMizFileObj.Out_AdjectiveList(aCluster: PList);
   var i: integer;
   begin Out_XElStart(XMLElemName[elAdjectiveCluster]);
   if aCluster↑.Count = 0 then
        begin Out_XElEnd0; exit;
        end;
   Out_XAttrEnd;
   with aCluster↑ do
        for i ← 0 to Count − 1 do Out_Adjective(Items↑[i]);
   Out_XElEnd(XMLElemName[elAdjectiveCluster]);
   end;
```

Subsection 21.11.1. Emitting XML for types

```
1021. Writing the XML for a Mizar type.
StandardType = element Standard-Type {
  attribute nr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term*
}
StructureType = element Structure-Type {
  attribute nr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term*
ClusteredType = element Clustered-Type {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Adjective-Cluster,
  Type,
}
Type = StandardType | StructureType | ClusteredType
  define print\_arguments(\#) \equiv
          if nArgs \uparrow . Count = 0 then Out\_XElEnd0
          else begin Out_XAttrEnd; Out_TermList(nArgs); Out_XElEnd(TypeName[#]);
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_Type(aTyp: TypePtr);
  begin with aTyp\uparrow do
    case aTyp \uparrow .nTypeSort of
    wsStandardType: with StandardTypePtr(aTyp)\uparrow do
        begin Out_XElStart(TypeName[wsStandardType]);
        Out\_XIntAttr(XMLAttrName[atNr], nModeSymbol);
        if nMizarAppearance then Out\_XAttr(XMLAttrName[atSpellinq], ModeName[nModeSymbol]);
        Out\_PosAsAttrs(nTypePos); print\_arguments(wsStandardType);
        end;
    wsStructureType: with StructTypePtr(aTyp)\uparrow do
        begin Out_XElStart(TypeName[wsStructureType]);
        Out\_XIntAttr(XMLAttrName[atNr], nStructSymbol);
        if nMizarAppearance then
          Out\_XAttr(XMLAttrName[atSpelling], StructureName[nStructSymbol]);
        Out_PosAsAttrs(nTypePos); print_arguments(wsStructureType);
        end;
    wsClusteredType: with ClusteredTypePtr(aTyp)\uparrow do
        begin Out_XElStart(TypeName[wsClusteredType]); Out_PosAsAttrs(nTypePos); Out_XAttrEnd;
        Out\_AdjectiveList(nAdjectiveCluster); Out\_Type(nType);
        Out\_XElEnd(TypeName[wsClusteredType]);
        end;
    wsErrorType: \mathbf{begin}\ Out\_XElWithPos(TypeName[wsErrorType], nTypePos);
    endcases;
```

§1021 Mizar Parser 331 EMITTING XML FOR TYPES

```
end;
1022.
        Printing a variable as an XML element.
Variable = element Variable {
  attribute idnr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer }
}
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_Variable(aVar : VariablePtr);
  begin with aVar\uparrow do
    \mathbf{begin}\ Out\_XElStart(XMLElemName[elVariable]);\ Out\_XIntAttr(XMLAttrName[atIdNr],nIdent);
    if nMizarAppearance then Out\_XAttr(XMLAttrName[atSpelling], IdentRepr(nIdent));
    Out_PosAsAttrs(nVarPos); Out_XElEnd0
    end;
  end;
        Variables introduced using "reserve" are just printed out like any other variable.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_ReservedVariable(aVar : VariablePtr);
  begin Out_{-}Variable(aVar);
  end:
1024. Implicitly qualified variables (i.e., variables which are reserved with a type, then used in, e.g., a
quantified formula) are just variables appearing as children of an "implicitly qualified" XML element.
  VariableSegment |= element Implicitly-Qualified-Segment {
    attribute col { xsd:integer },
    attribute line { xsd:integer },
    Variable
  }
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_ImplicitlyQualifiedVariable(aSegm: ImplicitlyQualifiedSegmentPtr);
  \textbf{begin} \ Out\_XElStart(SegmentKindName[ikImplQualifiedSegm]); \ Out\_PosAsAttrs(aSegm \uparrow .nSegmPos);
  Out\_XAttrEnd; Out\_Variable(aSegm \uparrow .nIdentifier);
  Out_XElEnd(SegmentKindName[ikImplQualifiedSegm]);
  end;
```

EMITTING XML FOR TYPES Mizar Parser §1025

1025. Qualified variable segments are either implicitly qualified (hence we use the previous function) or explicitly qualified (which look like " $\langle variable\ list\rangle$ being $\langle type\rangle$ ").

332

Explicitly qualified segments are an XML element with two children (a "variables" XML element, and a "type" XML element).

```
VariableSegment |= element Explicitly-Qualified-Segment {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  element Variables { Variable* },
  Type
}
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_VariableSegment(aSegm: QualifiedSegmentPtr);
  var i: integer;
  begin case aSegm\uparrow.nSegmentSort of
  ikImplQualifiedSeqm: Out\_ImplicitlyQualifiedVariable(ImplicitlyQualifiedSeqmentPtr(aSeqm));
  ikExplQualifiedSegm: with ExplicitlyQualifiedSegmentPtr(aSegm) \uparrow do
      begin Out\_XElStart(SegmentKindName[ikExplQualifiedSegm]); Out\_PosAsAttrs(nSegmPos);
      Out_XAttrEnd; Out_XElStart0(XMLElemName[elVariables]);
      for i \leftarrow 0 to nIdentifiers \uparrow. Count - 1 do Out\_Variable(nIdentifiers \uparrow. Items \uparrow [i]);
      Out\_XElEnd(XMLElemName[elVariables]); Out\_Type(nType);
      Out_XElEnd(SegmentKindName[ikExplQualifiedSegm]);
      end;
  endcases:
  end;
1026. Private predicates have the XML schema
  Private-Predicate-Formula = element Private-Predicate-Formula {
    attribute idnr { xsd:integer },
    attribute spelling { text }?,
    attribute col { xsd:integer },
    attribute line { xsd:integer },
    Term-List?
  }
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure Out WSMizFileObj.Out_PrivatePredicativeFormula(aFrm: PrivatePredicativeFormulaPtr);
  begin with PrivatePredicativeFormulaPtr(aFrm)↑ do
    begin Out_XElStart(FormulaName[wsPrivatePredicateFormula]);
    Out\_XIntAttr(XMLAttrName[atIdNr], nPredIdNr);
    if nMizarAppearance then Out\_XAttr(XMLAttrName[atSpelling], IdentRepr(nPredIdNr));
    Out\_PosAsAttrs(nFormulaPos);
    if nArgs \uparrow . Count = 0 then Out\_XElEnd0
    else begin Out_XAttrEnd; Out_TermList(nArgs);
      Out_XElEnd(FormulaName[wsPrivatePredicateFormula]);
      end;
    end;
  end;
```

Subsection 21.11.2. Emitting XML for formulas

Mizar Parser

1027. The XML schema for formulas looks something like:

```
Formula = NegatedFormula
 ConjunctiveFormula
 DisjunctiveFormula
 ConditionalFormula
 BiconditionalFormula
 FlexaryConjunctiveFormula
 FlexaryDisjunctiveFormula
 Predicative-Formula
 RightSideOf-Predicative-Formula
 Multi-Predicative-Formula
 Attributive-Formula
 Qualifying-Formula
 Universal-Quantifier-Formula
 Existential-Quantifier-Formula
 element Contradiction {
    attribute col { xsd:integer },
    attribute line { xsd:integer } }
| element Thesis {
    attribute col { xsd:integer },
    attribute line { xsd:integer } }
| element Formula-Error {
    attribute col { xsd:integer },
    attribute line { xsd:integer } }
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_Formula(aFrm:FormulaPtr);
  var i: integer:
  begin case aFrm\uparrow.nFormulaSort of
  wsNegatedFormula: \langle Emit XML for negated formula (WSM) 1028 \rangle;
  wsConjunctiveFormula: \( \)Emit XML for conjunction (WSM) 1029 \\;
  wsDisjunctiveFormula: (Emit XML for disjunction (WSM) 1030);
  wsConditionalFormula: (Emit XML for conditional formula (WSM) 1031);
  wsBiconditionalFormula: (Emit XML for biconditional formula (WSM) 1032);
  wsFlexaryConjunctiveFormula: \( \) Emit XML for flexary-conjunction (WSM) 1033 \( \);
  wsFlexaryDisjunctiveFormula: \( \) Emit XML for flexary-disjunction (WSM) 1034 \( \);
  wsPredicativeFormula: \( \) Emit XML for predicative formula (WSM) 1035 \( \);
  wsRightSideOfPredicativeFormula: \( \) Emit XML for right-side of predicative formula (WSM) 1036 \( \);
  wsMultiPredicativeFormula: (Emit XML for multi-predicative formula (WSM) 1037);
  wsPrivatePredicateFormula: Out\_PrivatePredicativeFormula(PrivatePredicativeFormulaPtr(aFrm));
  wsAttributiveFormula: (Emit XML for attributive formula (WSM) 1038);
  wsQualifyingFormula: \( \)Emit XML for qualifying formula (WSM) 1039 \( \);
  wsUniversalFormula: \( \)Emit XML for universal formula (WSM) 1040 \( \);
  wsExistentialFormula: \( \)Emit XML for existential formula (WSM) 1041 \( \);
  wsContradiction: begin Out\_XElWithPos(FormulaName[wsContradiction], aFrm\uparrow.nFormulaPos);
  wsThesis: \mathbf{begin} \ Out\_XElWithPos(FormulaName[wsThesis], aFrm \uparrow .nFormulaPos);
  wsErrorFormula: begin Out\_XElWithPos(FormulaName[wsErrorFormula], aFrm↑.nFormulaPos);
    end;
  endcases;
  end;
```

```
1028
```

```
1028.
NegatedFormula = element Negated-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula
\langle Emit XML for negated formula (WSM) 1028\rangle \equiv
  \textbf{begin} \ Out\_XElStart(FormulaName[wsNegatedFormula]); \ Out\_PosAsAttrs(aFrm \uparrow .nFormulaPos);
  Out\_XAttrEnd; Out\_Formula(NegativeFormulaPtr(aFrm)\uparrow.nArg);
  Out_XElEnd(FormulaName[wsNegatedFormula]);
  end
This code is used in section 1027.
1029.
ConjunctiveFormula = element Conjunctive-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula,
  Formula
\langle \text{ Emit XML for conjunction (WSM) } 1029 \rangle \equiv
  \textbf{begin } \textit{Out\_XElStart}(FormulaName[wsConjunctiveFormula]); \textit{Out\_PosAsAttrs}(aFrm\uparrow.nFormulaPos); \\
  Out\_XAttrEnd; Out\_Formula(BinaryFormulaPtr(aFrm)\uparrow.nLeftArq);
  Out\_Formula(BinaryFormulaPtr(aFrm)\uparrow.nRightArg);
  Out_XElEnd(FormulaName[wsConjunctiveFormula]);
  end
This code is used in section 1027.
1030.
DisjunctiveFormula = element Disjunctive-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula,
  Formula
}
\langle \text{ Emit XML for disjunction (WSM) } 1030 \rangle \equiv
  begin Out\_XElStart(FormulaName[wsDisjunctiveFormula]); Out\_PosAsAttrs(aFrm<math>\uparrow.nFormulaPos);
  Out\_XAttrEnd; Out\_Formula(BinaryFormulaPtr(aFrm)\uparrow.nLeftArg);
  Out\_Formula(BinaryFormulaPtr(aFrm)\uparrow.nRightArg);
  Out_XElEnd(FormulaName[wsDisjunctiveFormula]);
This code is used in section 1027.
```

This code is used in section 1027.

```
1031.
ConditionalFormula = element Conditional-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula,
  Formula
}
\langle Emit XML for conditional formula (WSM) 1031 \rangle \equiv
  begin Out\_XElStart(FormulaName[wsConditionalFormula]); Out\_PosAsAttrs(aFrm<math>\uparrow.nFormulaPos);
  Out\_XAttrEnd; Out\_Formula(BinaryFormulaPtr(aFrm)\uparrow.nLeftArg);
  Out\_Formula(BinaryFormulaPtr(aFrm)\uparrow.nRightArg);
  Out\_XElEnd(FormulaName[wsConditionalFormula]);
  end
This code is used in section 1027.
1032.
BiconditionalFormula = element Biconditional-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula.
  Formula
}
\langle \text{Emit XML for biconditional formula (WSM) } 1032 \rangle \equiv
  begin Out\_XElStart(FormulaName[wsBiconditionalFormula]); Out\_PosAsAttrs(aFrm<math>\uparrow.nFormulaPos);
  Out\_XAttrEnd; Out\_Formula(BinaryFormulaPtr(aFrm)\uparrow.nLeftArg);
  Out\_Formula(BinaryFormulaPtr(aFrm)\uparrow.nRightArg);
  Out\_XElEnd(FormulaName[wsBiconditionalFormula]);
  end
This code is used in section 1027.
1033.
FlexaryConjunctiveFormula = element FlexaryConjunctive-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula,
  Formula
}
\langle Emit XML for flexary-conjunction (WSM) 1033\rangle \equiv
  begin Out_XElStart(FormulaName[wsFlexaryConjunctiveFormula]);
  Out\_PosAsAttrs(aFrm\uparrow.nFormulaPos); Out\_XAttrEnd;
  Out\_Formula(BinaryFormulaPtr(aFrm)\uparrow.nLeftArg);
  Out\_Formula(BinaryFormulaPtr(aFrm)\uparrow.nRightArg);
  Out_XElEnd(FormulaName[wsFlexaryConjunctiveFormula]);
  end
```

```
5 --
```

```
1034.
FlexaryDisjunctiveFormula = element FlexaryDisjunctive-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula,
  Formula
}
\langle Emit XML for flexary-disjunction (WSM) 1034\rangle \equiv
  begin Out_XElStart(FormulaName[wsFlexaryDisjunctiveFormula]);
  Out\_PosAsAttrs(aFrm\uparrow.nFormulaPos); Out\_XAttrEnd;
  Out\_Formula(BinaryFormulaPtr(aFrm)\uparrow.nLeftArg);
  Out\_Formula(BinaryFormulaPtr(aFrm)\uparrow.nRightArg);
  Out_XElEnd(FormulaName[wsFlexaryDisjunctiveFormula]);
  end
This code is used in section 1027.
1035.
Predicative-Formula = element Predicative-Formula {
  attribute nr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  element Arguments { Term-List? },
  element Arguments { Term-List? }
}
\langle\,{\rm Emit}\,\,{\rm XML} for predicative formula (WSM) \,1035\,\rangle\equiv
  with PredicativeFormulaPtr(aFrm)↑ do
    begin Out_XElStart(FormulaName[wsPredicativeFormula]);
    Out\_XIntAttr(XMLAttrName[atNr], nPredNr);
    if nMizarAppearance then Out\_XAttr(XMLAttrName[atSpelling], PredicateName[nPredNr]);
    Out_PosAsAttrs(nFormulaPos); Out_XAttrEnd;
    if nLeftArgs \uparrow. Count = 0 then Out\_XEl1(XMLElemName[elArguments])
    else begin Out_XElStart0(XMLElemName[elArguments]); Out_TermList(nLeftArgs);
      Out_XElEnd(XMLElemName[elArguments]);
      end:
    if nRightArgs \uparrow. Count = 0 then Out\_XEl1(XMLElemName[elArguments])
    else begin Out_XElStart0(XMLElemName[elArguments]); Out_TermList(nRightArgs);
      Out_XElEnd(XMLElemName[elArguments]);
      end;
    Out_XElEnd(FormulaName[wsPredicativeFormula]);
    end
This code is used in section 1027.
```

```
RightSideOf-Predicative-Formula = element RightSideOf-Predicative-Formula {
  attribute nr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  element Arguments { Term-List? }
\langle Emit XML for right-side of predicative formula (WSM) 1036\rangle \equiv
  with RightSideOfPredicativeFormulaPtr(aFrm) \uparrow do
    begin Out_XElStart(FormulaName[wsRightSideOfPredicativeFormula]);
    Out\_XIntAttr(XMLAttrName[atNr], nPredNr);
    if nMizarAppearance then Out\_XAttr(XMLAttrName[atSpelling], PredicateName[nPredNr]);
    Out_PosAsAttrs(nFormulaPos); Out_XAttrEnd;
    if nRightArgs \uparrow. Count = 0 then Out\_XEl1(XMLElemName[elArguments])
    else begin Out_XElStart0(XMLElemName[elArguments]); Out_TermList(nRightArgs);
      Out\_XElEnd(XMLElemName[elArguments]);
    Out\_XElEnd(FormulaName[wsRightSideOfPredicativeFormula])
    end
This code is used in section 1027.
1037.
Multi-Predicative-Formula = element Multi-Predicative-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula*
\langle \text{ Emit XML for multi-predicative formula (WSM) } 1037 \rangle \equiv
  with MultiPredicativeFormulaPtr(aFrm) \uparrow do
    begin Out_XElStart(FormulaName[wsMultiPredicativeFormula]);
    Out_PosAsAttrs(aFrm↑.nFormulaPos); Out_XAttrEnd;
    for i \leftarrow 0 to nScraps.Count - 1 do Out\_Formula(nScraps \uparrow .Items \uparrow [i]);
    Out\_XElEnd(FormulaName[wsMultiPredicativeFormula])
    end
This code is used in section 1027.
```

This code is used in section 1027.

```
1038.
Attributive-Formula = element Attributive-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term,
  Adjective-Cluster.element
}
\langle \text{Emit XML for attributive formula (WSM) } 1038 \rangle \equiv
  with AttributiveFormulaPtr(aFrm)↑ do
    \textbf{begin} \ Out\_XElStart(FormulaName[wsAttributiveFormula]);} \ Out\_PosAsAttrs(nFormulaPos);
    Out_XAttrEnd; Out_Term(nSubject); Out_AdjectiveList(nAdjectives);
    Out_XElEnd(FormulaName[wsAttributiveFormula]);
    end
This code is used in section 1027.
1039.
Qualifying-Formula = element Qualifying-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term,
  Type,
  Formula
\langle \text{ Emit XML for qualifying formula (WSM) } 1039 \rangle \equiv
  with QualifyingFormulaPtr(aFrm)↑ do
    \textbf{begin} \ Out\_XElStart(FormulaName[wsQualifyingFormula]); \ Out\_PosAsAttrs(nFormulaPos);
    Out_XAttrEnd; Out_Term(nSubject); Out_Type(nType);
    Out_XElEnd(FormulaName[wsQualifyingFormula]);
This code is used in section 1027.
1040.
Universal-Quantifier-Formula = element Universal-Quantifier-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Variable-Segment,
  Formula
}
\langle \text{Emit XML for universal formula (WSM) } 1040 \rangle \equiv
  with QuantifiedFormulaPtr(aFrm)\uparrow do
    \mathbf{begin}\ Out\_XElStart(FormulaName[wsUniversalFormula]);\ Out\_PosAsAttrs(nFormulaPos);
    Out\_XAttrEnd; Out\_VariableSegment(QuantifiedFormulaPtr(aFrm)\uparrow.nSegment);
    Out\_Formula(QuantifiedFormulaPtr(aFrm)\uparrow.nScope);
    Out_XElEnd(FormulaName[wsUniversalFormula]);
    end
```

```
Existential-Quantifier-Formula = element Existential-Quantifier-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Variable-Segment,
  Formula
}

⟨ Emit XML for existential formula (WSM) 1041⟩ ≡
  with QuantifiedFormulaPtr(aFrm)↑ do
  begin Out_XElStart(FormulaName[wsExistentialFormula]); Out_PosAsAttrs(nFormulaPos);
  Out_XAttrEnd; Out_VariableSegment(QuantifiedFormulaPtr(aFrm)↑.nSegment);
  Out_Formula(QuantifiedFormulaPtr(aFrm)↑.nScope);
  Out_XElEnd(FormulaName[wsExistentialFormula]);
  end
```

This code is used in section 1027.

Subsection 21.11.3. Emitting XML for Terms

1042. We begin with simple terms.

```
Term |= element Simple-Term {
   attribute idnr { xsd:integer },
   attribute spelling { text }?,
   attribute col { xsd:integer },
   attribute line { xsd:integer },
   attribute line { xsd:integer }
}

⟨Implementation for wsmarticle.pas 854⟩ +=
procedure OutWSMizFileObj.Out_SimpleTerm(aTrm:SimpleTermPtr);
   begin Out_XElStart(TermName[wsSimpleTerm]);
   Out_XIntAttr(XMLAttrName[atIdNr], aTrm↑.nIdent);
   if nMizarAppearance then Out_XAttr(XMLAttrName[atSpelling], IdentRepr(aTrm↑.nIdent));
   Out_PosAsAttrs(aTrm↑.nTermPos); Out_XElEnd0;
   end;
```

 $\S1043$ Mizar Parser EMITTING XML FOR TERMS 341

```
1043.
       Terms: Private functors.
Term |= element Private-Functor-Term {
  attribute idnr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  element Arguments { Term-List }?
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_PrivateFunctorTerm(aTrm: PrivateFunctorTermPtr);
  begin with PrivateFunctorTermPtr(aTrm)\uparrow do
    begin Out_XElStart(TermName[wsPrivateFunctorTerm]);
    Out_XIntAttr(XMLAttrName[atIdNr], nFunctorIdent);
    if nMizarAppearance then Out\_XAttr(XMLAttrName[atSpelling], IdentRepr(nFunctorIdent));
    Out\_PosAsAttrs(nTermPos);
    if nArgs\uparrow.Count = 0 then Out\_XElEnd0
    else begin Out_XAttrEnd; Out_TermList(nArqs); Out_XElEnd(TermName[wsPrivateFunctorTerm]);
      end;
    end;
  end;
1044.
       Terms: internal selectors.
Term |= element Internal-Selector-Term {
  attribute nr { text },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer }
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
{f procedure}\ OutWSMizFileObj.Out\_InternalSelectorTerm(aTrm:InternalSelectorTermPtr);
  begin with aTrm \uparrow do
    begin Out_XElStart(TermName[wsInternalSelectorTerm]);
    Out\_XIntAttr(XMLAttrName[atNr], nSelectorSymbol);
    if nMizarAppearance then Out\_XAttr(XMLAttrName[atSpellinq], SelectorName[nSelectorSymbol]);
    Out_PosAsAttrs(nTermPos); Out_XElEnd0;
    end;
  end;
```

```
1045.
       Terms: numerals, anaphoric "it", error.
Term |= element Numeral {
    attribute number { xsd:int },
    attribute col { xsd:integer },
    attribute line { xsd:integer }
Term |= element It-Term {
    attribute col { xsd:integer },
    attribute line { xsd:integer }
  }
Term |= element Error-Term { }
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_Term(aTrm : TermPtr);
  var i: integer;
  begin case aTrm\uparrow.nTermSort of
  wsPlaceholderTerm: (Emit XML for placeholder (WSM) 1046);
  wsSimpleTerm: Out_SimpleTerm(SimpleTermPtr(aTrm));
  wsNumeralTerm: begin; Out_XElStart(TermName[wsNumeralTerm]);
    Out\_XIntAttr(XMLAttrName[atNumber], NumeralTermPtr(aTrm)\uparrow.nValue);
    Out\_PosAsAttrs(aTrm \uparrow .nTermPos); Out\_XElEnd0;
    end:
  wsInfixTerm: \langle Emit XML for infix term (WSM) 1047 \rangle;
  wsCircumfixTerm: \( \text{Emit XML for circumfix term (WSM) 1048} \);
  wsPrivateFunctorTerm: Out\_PrivateFunctorTerm(PrivateFunctorTermPtr(aTrm));
  wsAggregate Term: (Emit XML for aggregate term (WSM) 1049);
  wsSelectorTerm: (Emit XML for selector term (WSM) 1050);
  wsInternalSelectorTerm: Out\_InternalSelectorTerm(InternalSelectorTermPtr(aTrm));
  wsForgetfulFunctorTerm: \( \) Emit XML for forgetful functor (WSM) 1051 \( \);
  wsInternalForgetfulFunctorTerm: (Emit XML for internal forgetful functor (WSM) 1052);
  wsFraenkelTerm: (Emit XML for Fraenkel term (WSM) 1053);
  wsSimpleFraenkelTerm: (Emit XML for simple Fraenkel term (WSM) 1054);
  wsQualificationTerm: (Emit XML for qualification term (WSM) 1055);
  wsExactlyTerm: \( \)Emit XML for exactly qualification term (WSM) 1056 \( \);
  wsGlobalChoiceTerm: (Emit XML for global choice term (WSM) 1057);
  wsItTerm: Out\_XElWithPos(TermName[wsItTerm], aTrm \uparrow .nTermPos);
  wsErrorTerm: Out_XEl1(TermName[wsErrorTerm]);
  endcases;
  end;
```

 $\S1046$ Mizar Parser EMITTING XML FOR TERMS 343

```
1046.
       Terms: placeholders.
Term |= element Placeholder-Term {
  attribute nr { text },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer }
}
\langle \text{ Emit XML for placeholder (WSM) } 1046 \rangle \equiv
  begin Out_XElStart(TermName[wsPlaceholderTerm]);
  Out\_XIntAttr(XMLAttrName[atNr], PlaceholderTermPtr(aTrm)\uparrow.nLocusNr);
  if nMizarAppearance then Out_XAttr(XMLAttrName[atSpelling],
        QuoteStrForXML(PlaceHolderName[PlaceholderTermPtr(aTrm)\uparrow.nLocusNr]));
  Out\_PosAsAttrs(aTrm\uparrow.nTermPos); Out\_XElEnd0;
  end
This code is used in section 1045.
1047. Terms: infixed.
Term |= element Infix-Term {
  attribute nr { text },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  element Arguments { Term-List? },
  element Arguments { Term-List? }
}
\langle \text{ Emit XML for infix term (WSM) } 1047 \rangle \equiv
  with InfixTermPtr(aTrm)\uparrow do
    begin Out_XElStart(TermName[wsInfixTerm]);
    Out\_XIntAttr(XMLAttrName[atNr], nFunctorSymbol);
    if nMizarAppearance then Out\_XAttr(XMLAttrName[atSpelling], FunctorName[nFunctorSymbol]);
    Out_PosAsAttrs(nTermPos); Out_XAttrEnd;
    if nLeftArgs \uparrow. Count = 0 then Out\_XEl1(XMLElemName[elArguments])
    else begin Out_XElStart0(XMLElemName[elArguments]); Out_TermList(nLeftArgs);
      Out\_XElEnd(XMLElemName[elArguments]);
      end:
    if nRightArgs \uparrow. Count = 0 then Out\_XEl1(XMLElemName[elArguments])
    else begin Out_XElStart0(XMLElemName[elArguments]); Out_TermList(nRightArgs);
      Out_XElEnd(XMLElemName[elArguments]);
      end;
    Out_XElEnd(TermName[wsInfixTerm]);
    end
This code is used in section 1045.
```

344 EMITTING XML FOR TERMS Mizar Parser $\S1048$

```
1048.
       Terms: brackets.
Term |= element Circumfix-Term {
  attribute nr { text },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  element Right-Circumflex-Symbol {
    attribute nr { text },
    attribute spelling { text }?,
    attribute col { xsd:integer },
    attribute line { xsd:integer }
  },
  element Arguments { Term-List? }
\langle \text{Emit XML for circumfix term (WSM) 1048} \rangle \equiv
  with CircumfixTermPtr(aTrm)\uparrow do
    begin Out_XElStart(TermName[wsCircumfixTerm]);
    Out\_XIntAttr(XMLAttrName[atNr], nLeftBracketSymbol);
    if nMizarAppearance then
      Out\_XAttr(XMLAttrName[atSpelling], LeftBracketName[nLeftBracketSymbol]);
    Out\_PosAsAttrs(nTermPos); Out\_XAttrEnd;
    Out_XElStart(XMLElemName[elRightCircumflexSymbol]);
    Out\_XIntAttr(XMLAttrName[atNr], nRightBracketSymbol);
    if nMizarAppearance then
      Out\_XAttr(XMLAttrName[atSpelling], RightBracketName[nRightBracketSymbol]);
    Out_PosAsAttrs(nTermPos); Out_XElEnd0; Out_TermList(nArgs);
    Out_XElEnd(TermName[wsCircumfixTerm]);
    end
This code is used in section 1045.
1049.
       Terms: structure instances.
Term |= element Aggregate-Term {
  attribute nr { text },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  element Arguments { Term-List }?
}
\langle \text{ Emit XML for aggregate term (WSM) } 1049 \rangle \equiv
  with AggregateTermPtr(aTrm)\uparrow do
    begin Out_XElStart(TermName[wsAggregateTerm]);
    Out\_XIntAttr(XMLAttrName[atNr], nStructSymbol);
    if nMizarAppearance then Out\_XAttr(XMLAttrName[atSpelling], StructureName[nStructSymbol]);
    Out\_PosAsAttrs(nTermPos);
    if nArgs\uparrow.Count = 0 then Out\_XElEnd0
    else begin Out\_XAttrEnd; Out\_TermList(nArgs); Out\_XElEnd(TermName[wsAggregateTerm]);
      end:
    end
This code is used in section 1045.
```

1050. Terms: selectors.

EMITTING XML FOR TERMS 345

```
Term |= element Selector-Term {
  attribute nr { text },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term
}
\langle \text{Emit XML for selector term (WSM) } 1050 \rangle \equiv
  with SelectorTermPtr(aTrm)\uparrow do
    begin Out_XElStart(TermName[wsSelectorTerm]);
    Out\_XIntAttr(XMLAttrName[atNr], nSelectorSymbol);
    if nMizarAppearance then Out\_XAttr(XMLAttrName[atSpelling], SelectorName[nSelectorSymbol]);
    Out\_PosAsAttrs(nTermPos); Out\_XAttrEnd; Out\_Term(nArg);
    Out_XElEnd(TermName[wsSelectorTerm]);
    \mathbf{end}
This code is used in section 1045.
1051. Terms: forgetful functors.
Term |= element Forgetful-Functor-Term {
  attribute nr { text },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term
\langle Emit XML for forgetful functor (WSM) 1051\rangle \equiv
  with ForgetfulFunctorTermPtr(aTrm)\uparrow do
    begin Out_XElStart(TermName[wsForgetfulFunctorTerm]);
    Out\_XIntAttr(XMLAttrName[atNr], nStructSymbol);
    if nMizarAppearance then Out\_XAttr(XMLAttrName[atSpellinq], StructureName[nStructSymbol]);
    Out_PosAsAttrs(nTermPos); Out_XAttrEnd; Out_Term(nArg);
    Out_XElEnd(TermName[wsForgetfulFunctorTerm]);
    end
This code is used in section 1045.
        Terms: internal forgetful functors.
Term |= element Internal-Forgetful-Functor-Term {
  attribute nr { text },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
\langle \text{Emit XML for internal forgetful functor (WSM) } 1052 \rangle \equiv
  with InternalForgetfulFunctorTermPtr(aTrm) \uparrow do
    begin Out_XElStart(TermName[wsInternalForgetfulFunctorTerm]);
    Out\_XIntAttr(XMLAttrName[atNr], nStructSymbol); Out\_PosAsAttrs(nTermPos); Out\_XElEnd0;
    end
This code is used in section 1045.
```

§1050

Mizar Parser

§1053

Mizar Parser

This code is used in section 1045.

1053.

Terms: Fraenkel operators.

```
Term |= element Fraenkel-Term {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Variable-Segment*,
  Term,
  Formula
\langle \text{Emit XML for Fraenkel term (WSM) } 1053 \rangle \equiv
  with FraenkelTermPtr(aTrm)↑ do
    begin Out_XElStart(TermName[wsFraenkelTerm]); Out_PosAsAttrs(nTermPos); Out_XAttrEnd;
    for i \leftarrow 0 to nPostqualification \uparrow. Count - 1 do Out\_VariableSegment(nPostqualification \uparrow. Items \uparrow [i]);
    Out\_Term(nSample); Out\_Formula(nFormula); Out\_XElEnd(TermName[wsFraenkelTerm]);
    end
This code is used in section 1045.
      Terms: Simple Fraenkel expressions.
Term |= element Simple-Fraenkel-Term {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Variable-Segment*,
  Term
}
\langle \text{Emit XML for simple Fraenkel term (WSM) } 1054 \rangle \equiv
  with SimpleFraenkelTermPtr(aTrm) \uparrow do
    begin Out_XElStart(TermName[wsSimpleFraenkelTerm]); Out_PosAsAttrs(nTermPos);
    Out_XAttrEnd;
    for i \leftarrow 0 to nPostqualification \uparrow. Count - 1 do Out\_VariableSegment(nPostqualification \uparrow. Items \uparrow [i]);
    Out\_Term(nSample); Out\_XElEnd(TermName[wsSimpleFraenkelTerm]);
    end
This code is used in section 1045.
1055. Terms: qualification.
Term |= element Qualification-Term {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term,
  Type
\langle \text{Emit XML for qualification term (WSM) } 1055 \rangle \equiv
  with Qualified TermPtr(aTrm) \uparrow do
    \textbf{begin} \ Out\_XElStart(TermName[wsQualificationTerm]); \ Out\_PosAsAttrs(nTermPos); \ Out\_XAttrEnd;
    Out\_Term(nSubject); Out\_Type(nQualification); Out\_XElEnd(TermName[wsQualificationTerm]);
    end
```

EMITTING XML FOR TERMS 347

```
1056.
        Terms: exactly qualified.
Term |= element Exactly-Qualification-Term {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term
\langle Emit XML for exactly qualification term (WSM) 1056\rangle \equiv
  with ExactlyTermPtr(aTrm)\uparrow do
    begin Out_XElStart(TermName[wsQualificationTerm]); Out_PosAsAttrs(nTermPos);
    Out_XAttrEnd; Out_Term(nSubject); Out_XElEnd(TermName[wsQualificationTerm]);
    end
This code is used in section 1045.
1057. Terms: global choice expressions.
Term |= element Global-Choice-Term {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Туре
}
\langle Emit XML for global choice term (WSM) 1057\rangle \equiv
  begin Out\_XElStart(TermName[wsGlobalChoiceTerm]); Out\_PosAsAttrs(aTrm<math>\uparrow.nTermPos);
  Out\_XAttrEnd; Out\_Type(ChoiceTermPtr(aTrm)\uparrow.nChoiceType);
  Out_XElEnd(TermName[wsGlobalChoiceTerm]);
  end
This code is used in section 1045.
Subsection 21.11.4. Emitting XML for text items
1058. Type-lists are needed for text items.
Type-List = element Type-List {
  Type*
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_TypeList(aTypeList : PList);
  var i: integer;
  begin Out_XElStart0 (XMLElemName[elTypeList]);
  for i \leftarrow 0 to aTypeList \uparrow . Count - 1 do Out\_Type(aTypeList \uparrow . Items \uparrow [i]);
  Out\_XElEnd(XMLElemName[elTypeList]);
```

§1056

end;

Mizar Parser

```
1059. Locus.
Locus = element Locus {
  attribute idnr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer }
}
\langle Implementation for wsmarticle.pas 854\rangle +=
procedure OutWSMizFileObj.Out_Locus(aLocus: LocusPtr);
  begin with aLocus \uparrow do
    begin Out_XElStart(XMLElemName[elLocus]); Out_XIntAttr(XMLAttrName[atIdNr], nVarId);
    if nMizarAppearance then Out\_XAttr(XMLAttrName[atSpelling], IdentRepr(nVarId));
    Out\_PosAsAttrs(nVarIdPos); Out\_XElEnd0
    end;
  end;
1060.
Loci = element Loci { Locus* }
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_Loci(aLoci : PList);
  var i: integer;
  begin if (aLoci = nil) \lor (aLoci \uparrow. Count = 0) then Out\_XEl1(XMLElemName[elLoci])
  else begin Out_XElStart0(XMLElemName[elLoci]);
    for i \leftarrow 0 to aLoci \uparrow. Count - 1 do Out\_Locus(aLoci \uparrow. Items \uparrow [i]);
    Out\_XElEnd(XMLElemName[elLoci]);
    end;
  end;
1061. Patterns.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_Pattern(aPattern: PatternPtr);
  begin case aPattern \uparrow .nPatternSort of
  itDefPred: (Emit XML for predicate pattern (WSM) 1062);
  itDefFunc: begin case FunctorPatternPtr(aPattern) \underline{\cap-n}.nFunctKind of
    InfixFunctor: (Emit XML for infix functor pattern (WSM) 1063);
    CircumfixFunctor: (Emit XML for bracket functor pattern (WSM) 1064);
    endcases:
    end;
  itDefMode: (Emit XML for mode pattern (WSM) 1065);
itDefAttr: (Emit XML for attribute pattern (WSM) 1066);
  endcases;
  end;
```

```
1062.
```

```
Predicate-Pattern = element Predicate-Pattern {
  attribute nr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Loci,
  Loci
}
\langle \text{Emit XML for predicate pattern (WSM) } 1062 \rangle \equiv
  with PredicatePatternPtr(aPattern)↑ do
    begin Out_XElStart(DefPatternName[itDefPred]);
    Out\_XIntAttr(XMLAttrName[atNr], nPredSymbol);
    if nMizarAppearance then Out\_XAttr(XMLAttrName[atSpelling], PredicateName[nPredSymbol]);
    Out_PosAsAttrs(nPatternPos); Out_XAttrEnd; Out_Loci(nLeftArgs); Out_Loci(nRightArgs);
    Out_XElEnd(DefPatternName[itDefPred]);
    end
This code is used in section 1061.
1063.
Operation-Functor-Pattern = element Operation-Functor-Pattern {
  attribute nr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Loci,
  Loci
\langle \text{ Emit XML for infix functor pattern (WSM) } 1063 \rangle \equiv
  with InfixFunctorPatternPtr(aPattern)↑ do
    begin Out_XElStart(FunctorPatternName[InfixFunctor]);
    Out\_XIntAttr(XMLAttrName[atNr], nOperSymb);
    if nMizarAppearance then Out\_XAttr(XMLAttrName[atSpellinq], FunctorName[nOperSymb]);
    Out_PosAsAttrs(nPatternPos); Out_XAttrEnd; Out_Loci(nLeftArgs); Out_Loci(nRightArgs);
    Out_XElEnd(FunctorPatternName[InfixFunctor]);
    end
This code is used in section 1061.
```

```
Bracket-Functor-Pattern = element Bracket-Functor-Pattern {
  attribute nr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  element RightCircumflexSymbol {
    attribute nr { xsd:integer },
    attribute spelling { text }?
  },
  Loci
}
\langle Emit XML for bracket functor pattern (WSM) 1064\rangle \equiv
  with CircumfixFunctorPatternPtr(aPattern)↑ do
    begin Out_XElStart(FunctorPatternName[CircumfixFunctor]);
    Out\_XIntAttr(XMLAttrName[atNr], nLeftBracketSymb);
    if nMizarAppearance then
      Out\_XAttr(XMLAttrName[atSpelling], LeftBracketName[nLeftBracketSymb]);
    Out\_PosAsAttrs(nPatternPos); Out\_XAttrEnd;
    Out_XElStart(XMLElemName[elRightCircumflexSymbol]);
    Out\_XIntAttr(XMLAttrName[atNr], nRightBracketSymb);
    if nMizarAppearance then
      Out\_XAttr(XMLAttrName[atSpellinq], RightBracketName[nRightBracketSymb]);
    Out\_XAttrEnd; Out\_XElEnd(XMLElemName[elRightCircumflexSymbol]); Out\_Loci(nArgs);
    Out_XElEnd(FunctorPatternName[CircumfixFunctor]);
    end
This code is used in section 1061.
1065.
Mode-Pattern = element Mode-Pattern {
  attribute nr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Loci
}
\langle \text{ Emit XML for mode pattern (WSM) } 1065 \rangle \equiv
  with ModePatternPtr(aPattern)↑ do
    begin Out_XElStart(DefPatternName[itDefMode]);
    Out\_XIntAttr(XMLAttrName[atNr], nModeSymbol);
    if nMizarAppearance then Out\_XAttr(XMLAttrName[atSpelling], ModeName[nModeSymbol]);
    Out_PosAsAttrs(nPatternPos); Out_XAttrEnd; Out_Loci(nArgs);
    Out\_XElEnd(DefPatternName[itDefMode])
This code is used in section 1061.
```

1066. I am confused why there is both a locus and loci elements in an attribute pattern.

```
Attribute-Pattern = element Attribute-Pattern {
  attribute nr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Locus,
  Loci
}
\langle \text{Emit XML for attribute pattern (WSM) 1066} \rangle \equiv
  with AttributePatternPtr(aPattern) \uparrow do
    \textbf{begin} \ Out\_XElStart(DefPatternName[itDefAttr]); \ Out\_XIntAttr(XMLAttrName[atNr], nAttrSymbol);
    if nMizarAppearance then Out\_XAttr(XMLAttrName[atSpelling], AttributeName[nAttrSymbol]);
    Out_PosAsAttrs(nPatternPos); Out_XAttrEnd; Out_Locus(nArg); Out_Loci(nArgs);
    Out\_XElEnd(DefPatternName[itDefAttr]);
    end
This code is used in section 1061.
1067.
Label = element Label {
  attribute idnr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Locus,
  Loci
}
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_Label(aLab : LabelPtr);
    begin
    if (aLab \neq nil) { \land (aLab.nLabelIdNr > 0) }
    then
    begin Out_XElStart(XMLElemName[elLabel]);
    Out\_XIntAttr(XMLAttrName[atIdNr], aLab\uparrow.nLabelIdNr);
    if nMizarAppearance then Out\_XAttr(XMLAttrName[atSpelling], IdentRepr(aLab\uparrow.nLabelIdNr));
    Out\_PosAsAttrs(aLab\uparrow.nLabelPos); Out\_XElEnd0
    end;
    end;
```

case nExprKind of

 $exTerm: Out_Term(TermPtr(nExpr));$

 $exFormula: Out_Formula(FormulaPtr(nExpr));$

1068. Emitting XML for definiens. Definiens = element Definiens { attribute kind { "Simple-Definiens" }, attribute shape { text }?, Label, (Term | Formula) } | element Definiens { attribute kind { "Conditional-Definiens" }, attribute shape { text }?, Label, element Partial-Definiens { (Term | Formula)* }, (Term | Formula)? \langle Implementation for wsmarticle.pas $854 \rangle + \equiv$ **procedure** OutWSMizFileObj.Out_Definiens(aDef : DefiniensPtr); var i: integer; lExprKind: ExpKind; begin if $aDef \neq nil$ then with $DefiniensPtr(aDef)\uparrow do$ $begin Out_XElStart(XMLElemName[elDefiniens]); Out_PosAsAttrs(nDefPos);$ case nDefSort of SimpleDefiniens: with SimpleDefiniensPtr(aDef) \uparrow , nExpression \uparrow do **begin** Out_XAttr(XMLAttrName[atKind], DefiniensKindName[SimpleDefiniens]); Out_XAttr(XMLAttrName[atShape], ExpName[nExprKind]); Out_XAttrEnd; $Out_Label(nDefLabel);$ case nExprKind of $exTerm: Out_Term(TermPtr(nExpr));$ $exFormula: Out_Formula(FormulaPtr(nExpr));$ endcases; end: Conditional Definiens: with Conditional Definiens $Ptr(aDef) \uparrow do$ **begin** $Out_XAttr(XMLAttrName[atKind], DefiniensKindName[ConditionalDefiniens]);$ $lExprKind \leftarrow exFormula;$ if $nOtherwise \neq nil then lExprKind \leftarrow nOtherwise \uparrow .nExprKind$ else if $nConditionalDefiniensList\uparrow.Count > 0$ then $lExprKind \leftarrow$ $PartDefPtr(nConditionalDefiniensList\uparrow.Items\uparrow[0])\uparrow.nPartDefiniens\uparrow.nExprKind;$ $Out_XAttr(XMLAttrName[atShape], ExpName[lExprKind]); Out_XAttrEnd;$ $Out_Label(nDefLabel);$ for $i \leftarrow 0$ to $nConditionalDefiniensList \uparrow. Count - 1$ do with $PartDefPtr(nConditionalDefiniensList\uparrow.Items\uparrow[I])\uparrow do$ **begin** Out_XElStart0 (XMLElemName [elPartialDefiniens]); with nPartDefiniens↑ do case nExprKind of $exTerm: Out_Term(TermPtr(nExpr));$ $exFormula: Out_Formula(FormulaPtr(nExpr));$ endcases; $Out_Formula(nGuard); Out_XElEnd(XMLElemName[elPartialDefiniens]);$ if $nOtherwise \neq nil$ then with $nOtherwise \uparrow do$

```
endcases;
           end;
      endcases; Out_XElEnd(XMLElemName[elDefiniens]);
  end;
1069.
Proposition = element Proposition {
  Label,
  Formula
}
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_Proposition(aProp : PropositionPtr);
  begin Out\_XElStart(XMLElemName[elProposition]); Out\_XAttrEnd; Out\_Label(aProp<math>\uparrow.nLab);
  Out\_Formula(aProp \uparrow .nSentence); Out\_XElEnd(XMLElemName[elProposition]);
  end;
1070.
Local-Reference = element Local-Reference {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  attribute idnr { xsd:integer },
  attribute spelling { text }?
}
\langle Implementation for wsmarticle.pas 854\rangle + \equiv
procedure OutWSMizFileObj.Out_LocalReference(aRef : LocalReferencePtr);
  begin with LocalReferencePtr(aRef)\uparrow do
    \textbf{begin} \ \ Out\_XElStart(ReferenceKindName[LocalReference]); \ \ Out\_PosAsAttrs(nRefPos); \\
    Out\_XIntAttr(XMLAttrName[atIdNr], nLabId);
    if nMizarAppearance then Out\_XAttr(XMLAttrName[atSpelling], IdentRepr(nLabId));
    Out\_XElEnd0;
    end;
  end;
```

354

```
References = (Local-Reference
  | element Theorem-Reference {
    attribute col { xsd:integer },
    attribute line { xsd:integer },
    attribute at { xsd:integer },
    attribute spelling { text }?,
    attribute nr { xsd:integer }
} | element Definition-Reference {
    attribute col { xsd:integer },
    attribute line { xsd:integer },
    attribute at { xsd:integer },
    attribute spelling { text }?,
     attribute nr { xsd:integer }
})*
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_References(aRefs: PList);
  var i: integer;
  begin for i \leftarrow 0 to aRefs \uparrow . Count - 1 do
    with ReferencePtr(aRefs\uparrow.Items\uparrow[i])\uparrow do
       case nRefSort of
       LocalReference: Out\_LocalReference(aRefs\uparrow.Items\uparrow[i]);
       TheoremReference: begin Out_XElStart(ReferenceKindName[TheoremReference]);
         Out\_PosAsAttrs(nRefPos);
         Out\_XIntAttr(XMLAttrName[atNr], TheoremReferencePtr(aRefs\uparrow.Items\uparrow[i])\uparrow.nArticleNr);
         if nMizarAppearance then Out_XAttr(XMLAttrName[atSpelling],
                MMLIdentifierName[TheoremReferencePtr(aRefs\uparrow.Items\uparrow[i])\uparrow.nArticleNr]);
         Out\_XIntAttr(XMLAttrName[atNumber], TheoremReferencePtr(aRefs\uparrow.Items\uparrow[i])\uparrow.nTheoNr);
         Out\_XElEnd0;
         end:
       DefinitionReference: begin Out_XElStart(ReferenceKindName[DefinitionReference]);
         Out\_PosAsAttrs(nRefPos);
         Out\_XIntAttr(XMLAttrName[atNr], DefinitionReferencePtr(aRefs\uparrow.Items\uparrow[i])\uparrow.nArticleNr);
         if nMizarAppearance then Out\_XAttr(XMLAttrName[atSpelling],
                MMLIdentifierName [TheoremReferencePtr(aRefs\uparrow.Items\uparrow[i])\uparrow.nArticleNr]);
         Out\_XIntAttr(XMLAttrName[atNumber], DefinitionReferencePtr(aRefs\uparrow.Items\uparrow[i])\uparrow.nDefNr);
         Out\_XElEnd0;
         end:
       endcases;
  end:
```

```
§1072
        Mizar Parser
                                                              EMITTING XML FOR TEXT ITEMS
1072.
Link = element Link {
  attribute col { xsd:integer },
  attribute line { xsd:integer }
}
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_Link(aInf : JustificationPtr);
  begin with StraightforwardJustificationPtr(aInf) \uparrow \mathbf{do}
    if nLinked then
      begin Out_XElStart(XMLElemName[elLink]); Out_PosAsAttrs(nLinkPos); Out_XElEnd0;
      end;
  end;
1073.
Scheme-Justification = element Scheme-Justification {
  attribute nr { xsd:integer },
  attribute idnr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  attribute poscol { xsd:integer },
  attribute posline { xsd:integer },
  References
}
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_SchemeJustification(aInf:SchemeJustificationPtr);
  begin with aInf↑ do
    begin Out_XElStart(InferenceName[infSchemeJustification]);
    Out\_XIntAttr(XMLAttrName[atNr], nSchFileNr);
    Out\_XIntAttr(XMLAttrName[atIdNr], nSchemeIdNr);
    if nMizarAppearance then
      if nSchFileNr > 0 then Out\_XAttr(XMLAttrName[atSpelling], MMLIdentifierName[nSchFileNr])
      else if nSchemeIdNr > 0 then Out\_XAttr(XMLAttrName[atSpelling], IdentRepr(nSchemeIdNr));
    Out\_PosAsAttrs(nInfPos); Out\_XIntAttr(XMLAttrName[atPosLine], nSchemeInfPos.Line);
    Out\_XIntAttr(XMLAttrName[atPosCol], nSchemeInfPos.Col); Out\_XAttrEnd;
    Out_References(nReferences); Out_XElEnd(InferenceName[infSchemeJustification]);
    end;
  end;
```

```
Justification =
( element Straightforward-Justification {
    attribute col { xsd:integer },
    attribute line { xsd:integer },
     (Link, References)?
  }
 Scheme-Justification
 element Inference-Error {
    attribute col { xsd:integer },
    attribute line { xsd:integer }
| element Skipped-Proof {
    attribute col { xsd:integer },
    attribute line { xsd:integer }
 Block # proof block
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_Justification(aInf: JustificationPtr; aBlock: wsBlockPtr);
  begin case aInf \uparrow .nInfSort of
  infStraightforwardJustification: with StraightforwardJustificationPtr(aInf) \uparrow do
      begin Out\_XElStart(InferenceName[infStraightforwardJustification]); Out\_PosAsAttrs(nInfPos);
      if \neg nLinked \land (nReferences \uparrow. Count = 0) then Out\_XElEndO
      else begin Out_XAttrEnd; Out_Link(aInf); Out_References(nReferences);
         Out_XElEnd(InferenceName[infStraightforwardJustification]);
        end:
      end:
  infSchemeJustification: Out\_SchemeJustification(SchemeJustificationPtr(aInf));
  infError: Out\_XElWithPos(InferenceName[infError], aInf \uparrow.nInfPos);
  infSkippedProof: Out\_XElWithPos(InferenceName[infSkippedProof], aInf \uparrow .nInfPos);
  infProof: Out_Block(aBlock);
  endcases:
  end;
1075.
Compact-Statement = (Proposition, Justification)
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_CompactStatement(aCStm : CompactStatementPtr;
         aBlock: wsBlockPtr);
  begin with aCStm\uparrow do
    begin Out_Proposition(nProp); Out_Justification(nJustification, aBlock);
    end:
  end;
```

```
§1076
                                                                                                         357
         Mizar Parser
                                                                    EMITTING XML FOR TEXT ITEMS
1076.
Regular-Statement =
( (Label, Block)
 Compact-Statement
(Compact-Statement,
   element Iterative-Step {
      attribute col { xsd:integer },
      attribute line { xsd:integer },
      Term,
      Justification
   })*
)
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_RegularStatement(aRStm: RegularStatementPtr; aBlock: wsBlockPtr);
  var i: integer;
  begin case aRStm\uparrow.nStatementSort of
  stDiffuseStatement: begin Out\_Label(DiffuseStatementPtr(aRStm)\uparrow.nLab); Out\_Block(aBlock);
  stCompactStatement: Out\_CompactStatement(CompactStatementPtr(aRStm), aBlock);
  stIterativeEquality: \mathbf{begin}\ Out\_CompactStatement(CompactStatementPtr(aRStm), \mathbf{nil});
    with IterativeEqualityPtr(aRStm)\uparrow do
      for i \leftarrow 0 to nIterSteps \uparrow. Count - 1 do
         with IterativeStepPtr(nIterSteps\uparrow.Items\uparrow[i])\uparrow do
           begin Out_XElStart(XMLElemName[elIterativeStep]); Out_PosAsAttrs(nIterPos);
           Out_XAttrEnd; Out_Term(nTerm); Out_Justification(nJustification, nil);
           Out_XElEnd(XMLElemName[elIterativeStep]);
           end:
    end:
  endcases;
  end;
1077.
Variables = element Variables {
  Variable*
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_ReservationSegment(aRes: ReservationSegmentPtr);
  var i: integer;
  begin with aRes \uparrow do
    begin Out_XElStart0 (XMLElemName[elVariables]);
    for i \leftarrow 0 to nIdentifiers \uparrow. Count - 1 do Out\_Reserved Variable (nIdentifiers \uparrow. Items \uparrow [i]);
    Out\_XElEnd(XMLElemName[elVariables]); Out\_Type(nResType);
    end:
  end;
1078. \langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_SchemeNameInSchemeHead(aSch: SchemePtr);
  begin Out\_XIntAttr(XMLAttrName[atIdNr], aSch \uparrow .nSchemeIdNr);
```

if nMizarAppearance **then** $Out_XAttr(XMLAttrName[atSpelling], IdentRepr(aSch \cdot .nSchemeIdNr));$

end;

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_ItemContentsAttr(aWSItem: WSItemPtr);
  begin with aWSItem \uparrow do
    begin CurPos \leftarrow nItemPos;
    if nDisplayInformationOnScreen then DisplayLine(CurPos.Line, ErrorNbr);
    case nItemKind of
    itDefinition, itSchemeBlock, itSchemeHead, itTheorem, itAxiom, itReservation:;
    itSection::
    itConclusion, itRegularStatement: case RegularStatementPtr(nContent)\uparrow .nStatementSort of
      stDiffuseStatement:
             Out\_XAttr(XMLAttrName[atShape], RegularStatementName[stDiffuseStatement]);
      stCompactStatement:
             Out\_XAttr(XMLAttrName[atShape], RegularStatementName[stCompactStatement]);
      stIterative Equality: Out\_XAttr(XMLAttrName[atShape], Regular StatementName[stIterative Equality]);
      endcases;
    itChoice, itReconsider, itPrivFuncDefinition, itPrivPredDefinition, itConstantDefinition, itGeneralization,
           itLociDeclaration, itExistentialAssumption, itExemplification, itPerCases, itCaseBlock:;
    itCaseHead, itSupposeHead, itAssumption:;
    itCorrCond: Out_XAttr(XMLAttrName[atCondition],
           CorrectnessName[CorrectnessConditionPtr(nContent)\uparrow.nCorrCondSort]);
    itCorrectness: Out\_XAttr(XMLAttrName[atCondition], CorrectnessName[syCorrectness]);
    it Property:
           Out\_XAttr(XMLAttrName[atProperty], PropertyName[PropertyPtr(nContent)\uparrow.nPropertySort]);
    itDefFunc: Out_XAttr(XMLAttrName[atShape],
           Defining WayName [Functor Definition Ptr(nContent) \uparrow .nDefining Way]);
    itDefPred, itDefMode, itDefAttr, itDefStruct, itPredSynonym, itPredAntonym, itFuncNotation,
           itModeNotation, itAttrSynonym, itAttrAntonym, itCluster, itIdentify, itReduction:;
    it Property Registration:
           Out\_XAttr(XMLAttrName[atProperty], PropertyName[PropertyPtr(nContent)\uparrow.nPropertySort]);
    itPragma:
           Out\_XAttr(XMLAttrName[atSpelling], QuoteStrForXML(PragmaPtr(nContent)\uparrow.nPragmaStr));
    endcases;
    end;
  end;
```

1080. Emitting XML for item contents. This is used to expedite emitting the XML for a text-item (§1095).

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_ItemContents(aWSItem: WSItemPtr);
  var i, j: integer; s: CorrectnessKind;
  begin with aWSItem \uparrow do
    begin case nItemKind of
    itDefinition: Out_Block(nBlock);
    itSchemeBlock: Out_Block(nBlock);
    itSchemeHead: (Emit XML for schema (WSM) 1081);
    itTheorem: Out_CompactStatement(CompactStatementPtr(nContent), nBlock);
    itAxiom: begin end;
    itReservation: Out\_ReservationSegment(ReservationSegmentPtr(nContent));
    itSection:;
    itConclusion, itRegularStatement: Out\_RegularStatement(RegularStatementPtr(nContent), nBlock);
    itChoice: (Emit XML for consider contents (WSM) 1082);
    itReconsider: (Emit XML for reconsider contents (WSM) 1083);
    (Emit XML for definition-related items (WSM) 1084);
    itPredSynonym, itPredAntonym, itFuncNotation, itModeNotation, itAttrSynonym, itAttrAntonym:
            with NotationDeclarationPtr(nContent)↑ do
        begin Out_Pattern(nOriginPattern); Out_Pattern(nNewPattern);
        end;
    (Emit XML for registration-related items (WSM) 1093);
    itPragma:;
    itIncorrItem:;
    end;
    endcases;
  end;
```

```
Item-contents |= Scheme-contents
Scheme-contents = element Scheme {
  attribute idnr { xsd:integer },
  attribute spelling { text }?,
  element Schematic-Variables {
   (element Predicate-Segment {
       attribute col { xsd:integer },
       attribute line { xsd:integer },
       element Variables { Variable* },
    } | element Functor-Segment {
       attribute col { xsd:integer },
       attribute line { xsd:integer },
       element Variables { Variable* },
       Type-List,
       element Type-Specification { Type }
    })*
  },
  Formula,
  element Provisional-Formulas { Proposition* }?
\langle \text{ Emit XML for schema (WSM) } 1081 \rangle \equiv
  with SchemePtr(nContent) \uparrow do
    begin Out_XElStart(XMLElemName[elScheme]);
    Out\_SchemeNameInSchemeHead(SchemePtr(nContent)); Out\_XElEnd0;
    Out_XElStart0 (XMLElemName[elSchematicVariables]);
    for j \leftarrow 0 to nSchemeParams \uparrow. Count - 1 do
      case SchemeSegmentPtr(nSchemeParams.Items\uparrow[j])\uparrow.nSegmSort of
      PredicateSegment: with PredicateSegmentPtr(nSchemeParams.Items\uparrow[j])\uparrow do
           \mathbf{begin}\ Out\_XElStart(SchemeSegmentName[PredicateSegment]);\ Out\_PosAsAttrs(nSegmPos);
           Out_XAttrEnd; Out_XElStart0(XMLElemName[elVariables]);
           for i \leftarrow 0 to nVars \uparrow. Count - 1 do Out\_Variable(nVars.Items \uparrow [i]);
           Out\_XElEnd(XMLElemName[elVariables]); Out\_TypeList(nTypeExpList);
           Out\_XElEnd(SchemeSegmentName[PredicateSegment]);
           end:
      FunctorSegment: with FunctorSegmentPtr(nSchemeParams.Items\uparrow[j])\uparrow do
           \textbf{begin} \ Out\_XElStart(SchemeSegmentName[FunctorSegment]); \ Out\_PosAsAttrs(nSegmPos);
           Out_XAttrEnd; Out_XElStart0(XMLElemName[elVariables]);
           for i \leftarrow 0 to nVars \uparrow. Count - 1 do Out\_Variable(nVars.Items \uparrow [i]);
           Out\_XElEnd(XMLElemName[elVariables]); Out\_TypeList(nTypeExpList);
           Out\_XElStart0 (XMLElemName [elTypeSpecification]); Out\_Type (nSpecification);
           Out_XElEnd(XMLElemName[elTypeSpecification]);
           Out_XElEnd(SchemeSegmentName[FunctorSegment]);
           end;
      endcases:
    Out\_XElEnd(XMLElemName[elSchematicVariables]); Out\_Formula(nSchemeConclusion);
    if (nSchemePremises \neq nil) \land (nSchemePremises \uparrow. Count > 0) then
      begin Out_XElStart0 (XMLElemName[elProvisionalFormulas]);
      for i \leftarrow 0 to nSchemePremises \uparrow. Count - 1 do Out\_Proposition(nSchemePremises \uparrow. Items \uparrow [i]);
      Out_XElEnd(XMLElemName[elProvisionalFormulas]);
      end;
```

```
end
```

This code is used in section 1080.

This code is used in section 1080.

```
1082.
Item-contents |= Consider-Statement-contents
Consider-Statement-contents =
( Variable-Segment*,
  element Conditions { Proposition },
  Justification
)
\langle \text{Emit XML for consider contents (WSM) } 1082 \rangle \equiv
  with ChoiceStatementPtr(nContent) \uparrow do
    begin for i \leftarrow 0 to nQualVars \uparrow.Count - 1 do Out\_VariableSegment(nQualVars \uparrow.Items \uparrow [i]);
    Out_XElStart0(XMLElemName[elConditions]);
    for i \leftarrow 0 to nConditions \uparrow. Count - 1 do Out\_Proposition(nConditions \uparrow. Items \uparrow [i]);
    Out\_XElEnd(XMLElemName[elConditions]); Out\_Justification(nJustification, nil);
    end
This code is used in section 1080.
1083.
Item-contents |= Type-Changing-Statement-contents
Type-Changing-Statement-contents =
((element Equality {
    Variable,
    Term
  } | Variable),
 Type)
\langle \text{ Emit XML for reconsider contents (WSM) } 1083 \rangle \equiv
  with TypeChangingStatementPtr(nContent) \uparrow do
    begin for i \leftarrow 0 to nTypeChangeList \uparrow.Count - 1 do
       case TypeChangePtr(nTypeChangeList.Items\uparrow[i])\uparrow.nTypeChangeKind of
       Equating: begin Out_XElStart0(XMLElemName[elEquality]);
         Out\_Variable(TypeChangePtr(nTypeChangeList.Items\uparrow[i])\uparrow.nVar);
         Out\_Term(TypeChangePtr(nTypeChangeList.Items\uparrow[i])\uparrow.nTermExpr);
         Out_XElEnd(XMLElemName[elEquality]);
       Variable Identifier: \mathbf{begin} \ Out\_Variable (Type Change Ptr(nType Change List.Items \uparrow [i]) \uparrow. nVar);
         end:
       endcases;
    Out\_Type(nTypeExpr); Out\_Justification(nJustification, nil);
```

```
1084. We will need to recall Out_Variable (§1022) fr PrivateFunctorDefinitionObj (§893).
Item-contents |=
  (Variable, Type-List, Term) # private functors and predicates
 (Variable, Term)
                                    # constants
                                    # loci
 Variable-Segment
\langle \text{ Emit XML for definition-related items (WSM) } 1084 \rangle \equiv
itPrivFuncDefinition: with PrivateFunctorDefinitionPtr(nContent) \uparrow do
    begin Out_Variable(nFuncId); Out_TypeList(nTypeExpList); Out_Term(nTermExpr);
itPrivPredDefinition: with PrivatePredicateDefinitionPtr(nContent) \uparrow do
    begin Out_Variable(nPredId); Out_TypeList(nTypeExpList); Out_Formula(nSentence);
    end;
itConstantDefinition: with ConstantDefinitionPtr(nContent) \uparrow do
    begin Out_Variable(nVarId); Out_Term(nTermExpr);
    end;
itLociDeclaration, itGeneralization: Out\_VariableSegment(QualifiedSegmentPtr(nContent));
itCaseHead, itSupposeHead, itAssumption: (Emit XML for assumptions item (WSM) 1092);
See also sections 1085, 1086, 1087, 1088, 1089, 1090, and 1091.
This code is used in section 1080.
1085.
Item-contents |=
( Variable-Segment*,
  element Conditions { Proposition* } )
\langle Emit XML for definition-related items (WSM) 1084\rangle +\equiv
itExistentialAssumption: with ExistentialAssumptionPtr(nContent) \uparrow do
    begin for i \leftarrow 0 to nQVars \uparrow.Count - 1 do Out\_VariableSegment(nQVars \uparrow.Items \uparrow [i]);
    Out\_XElStart0 (XMLElemName [elConditions]);
    for i \leftarrow 0 to nConditions \uparrow. Count - 1 do Out\_Proposition(nConditions \uparrow. Items \uparrow [i]);
    Out\_XElEnd(XMLElemName[elConditions]);
    end;
1086.
Item-contents |= ( Variable?, Term? ) # Exemplification
                                              # percases, correctness-condition
                    Justification
                   Block
                                              # case block
\langle Emit XML for definition-related items (WSM) 1084\rangle +=
itExemplification: with ExamplePtr(nContent) \uparrow do
    begin if nVarId \neq nil then Out\_Variable(nVarId);
    if nTermExpr \neq nil then Out\_Term(nTermExpr);
itPerCases: Out\_Justification(JustificationPtr(nContent), \mathbf{nil});
itCaseBlock: Out_Block(nBlock);
itCorrCond: Out\_Justification(CorrectnessConditionPtr(nContent) \uparrow .nJustification, nBlock);
```

3-00.

```
1087.
Item-contents |=
  element CorrectnessConditions { # sic!
    element Correctness { attribute condition { text } }*,
       Justification }
Justification # Property
\langle Emit XML for definition-related items (WSM) 1084\rangle +=
itCorrectness: begin Out_XElStart0(XMLElemName[elCorrectnessConditions]);
  for s \in CorrectnessConditionsPtr(nContent) \uparrow .nConditions do
    begin Out_XElStart(ItemName[itCorrectness]);
    Out\_XAttr(XMLAttrName[atCondition], CorrectnessName[s]); Out\_XElEnd0;
    end:
  Out_XElEnd(XMLElemName[elCorrectnessConditions]);
  Out\_Justification(CorrectnessPtr(nContent)\uparrow.nJustification, nBlock);
itProperty: Out\_Justification(PropertyPtr(nContent) \uparrow .nJustification, nBlock);
1088.
Item-contents |=
( element Redefine { }?,
  Pattern,
  element Standard-Mode { Type },
  | element Expandable-Mode {
      element Type-Specification { Type }?,
      Definiens
    })
\langle Emit XML for definition-related items (WSM) 1084\rangle +\equiv
itDefMode: with ModeDefinitionPtr(nContent) \uparrow do
    begin if nRedefinition then Out_XEl1(XMLElemName[elRedefine]);
    Out\_Pattern(nDefModePattern);
    case nDefKind of
    defExpandableMode: begin Out\_XElStartO(ModeDefinitionSortName[defExpandableMode]);
      Out\_Type(ExpandableModeDefinitionPtr(nContent)\uparrow.nExpansion);
      Out\_XElEnd(ModeDefinitionSortName[defExpandableMode]);
    defStandardMode: with StandardModeDefinitionPtr(nContent) \uparrow do
        begin Out_XElStart0 (ModeDefinitionSortName[defStandardMode]);
        if nSpecification \neq nil then
          begin Out_XElStart0 (XMLElemName[elTypeSpecification]); Out_Type(nSpecification);
           Out_XElEnd(XMLElemName[elTypeSpecification]);
        Out_Definiens(nDefiniens); Out_XElEnd(ModeDefinitionSortName[defStandardMode]);
    endcases;
    end;
```

Mizar Parser

end;

EMITTING XML FOR TEXT ITEMS

1089. Item-contents |= (element Redefine { }?, Pattern, Definiens) \langle Emit XML for definition-related items (WSM) 1084 $\rangle + \equiv$ itDefAttr: with $AttributeDefinitionPtr(nContent) \uparrow$ do **begin if** nRedefinition **then** Out_XEl1 (XMLElemName[elRedefine]); Out_Pattern(nDefAttrPattern); Out_Definiens(nDefiniens); end: itDefPred: with $PredicateDefinitionPtr(nContent) \uparrow$ do **begin if** nRedefinition **then** Out_XEl1 (XMLElemName[elRedefine]); Out_Pattern(nDefPredPattern); Out_Definiens(nDefiniens); end; 1090. Item-contents |= (element Redefine { }?, Pattern, element Type-Specification { Type }?, Definiens) \langle Emit XML for definition-related items (WSM) 1084 \rangle += itDefFunc: with $FunctorDefinitionPtr(nContent) \uparrow do$ **begin if** nRedefinition **then** Out_XEl1(XMLElemName[elRedefine]); $Out_Pattern(nDefFuncPattern);$ if $nSpecification \neq nil$ then **begin** Out_XElStart0 (XMLElemName[elTypeSpecification]); Out_Type(nSpecification); Out_XElEnd(XMLElemName[elTypeSpecification]); $Out_Definiens(nDefiniens);$

1091.

```
Item-contents |=
(element Ancestors { Type* },
   attribute nr { xsd:integer },
   attribute spelling { text }?,
   attribute col { xsd:integer },
   attribute line { xsd:integer },
   Loci,
   (element Field-Segment {
       attribute col { xsd:integer },
       attribute line { xsd:integer },
       (element Selector {
         attribute nr { xsd:integer },
         attribute spelling { text }?,
         attribute col { xsd:integer },
         attribute line { xsd:integer }
       })*,
       Туре
   )*
 },
\langle Emit XML for definition-related items (WSM) 1084\rangle + \equiv
itDefStruct: with StructureDefinitionPtr(nContent) \uparrow do
    begin Out_XElStart0 (XMLElemName [elAncestors]);
    for i \leftarrow 0 to nAncestors \uparrow. Count - 1 do Out\_Type(nAncestors \uparrow. Items \uparrow [i]);
    Out\_XElEnd(XMLElemName[elAncestors]); Out\_XElStart(DefPatternName[itDefStruct]);
    Out\_XIntAttr(XMLAttrName[atNr], nDefStructPattern \uparrow.nModeSymbol);
    if nMizarAppearance then
       Out\_XAttr(XMLAttrName[atSpelling], StructureName[nDefStructPattern \uparrow.nModeSymbol]);
    Out\_PosAsAttrs(nStrPos); Out\_XAttrEnd; Out\_Loci(nDefStructPattern \uparrow .nArgs);
    for i \leftarrow 0 to nSgmFields \uparrow. Count - 1 do
       with FieldSegmentPtr(nSgmFields\uparrow.Items\uparrow[i])\uparrow do
         begin Out_XElStart(XMLElemName[elFieldSegment]); Out_PosAsAttrs(nFieldSegmPos);
         Out\_XAttrEnd;
         for j \leftarrow 0 to nFields \uparrow. Count - 1 do
           with FieldSymbolPtr(nFields\uparrow.Items\uparrow[j])\uparrow do
             begin Out_XElStart(XMLElemName[elSelector]);
              Out\_XIntAttr(XMLAttrName[atNr], nFieldSymbol);
             if nMizarAppearance then
                Out\_XAttr(XMLAttrName[atSpelling], SelectorName[nFieldSymbol]);
             Out\_PosAsAttrs(nFieldPos); Out\_XElEnd0
         Out_Type(nSpecification); Out_XElEnd(XMLElemName[elFieldSegment]);
    Out_XElEnd(DefPatternName[itDefStruct]);
    end
```

1092.

```
Item-contents |= (element Single-Assumption {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Proposition
} | element Collective-Assumption {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  element Conditions { Proposition* }
})
\langle \text{Emit XML for assumptions item (WSM) } 1092 \rangle \equiv
  case AssumptionPtr(nContent)\uparrow.nAssumptionSort of
  Single Assumption: \mathbf{begin} \ Out\_XElStart(AssumptionKindName[Single Assumption]);
    Out\_PosAsAttrs(AssumptionPtr(nContent)\uparrow.nAssumptionPos);\ Out\_XAttrEnd;
    Out\_Proposition(SingleAssumptionPtr(nContent)\uparrow.nProp);
    Out\_XElEnd(AssumptionKindName[SingleAssumption]);
    end;
  Collective Assumption: \mathbf{begin} \ Out\_XElStart(AssumptionKindName[Collective Assumption]);
    Out\_PosAsAttrs(AssumptionPtr(nContent)\uparrow.nAssumptionPos);\ Out\_XAttrEnd;
    Out\_XElStart0 (XMLElemName[elConditions]);
    with Collective Assumption Ptr(nContent) \uparrow do
      for i \leftarrow 0 to nConditions \uparrow. Count - 1 do Out\_Proposition(nConditions \uparrow. Items \uparrow [i]);
    Out\_XElEnd(XMLElemName[elConditions]);
    Out\_XElEnd(AssumptionKindName[CollectiveAssumption]);
    end:
  endcases
```

This code is used in section 1084.

1093. We have cluster registrations and non-cluster registrations.

```
Existential-Registration-content = element Existential-Registration {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Adjective-Cluster,
  Type
}
Conditional-Registration-content = element Conditional-Registration {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Adjective-Cluster, Adjective-Cluster,
  Туре
}
Functorial-Registration-content = element Functorial-Registration {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term,
  Adjective-Cluster,
  Type?
\langle Emit XML for registration-related items (WSM) 1093\rangle \equiv
itCluster: case ClusterPtr(nContent) \upsilon.nClusterKind of
  ExistentialRegistration: with EClusterPtr(nContent)\uparrow do
      begin Out_XElStart(ClusterRegistrationName[ExistentialRegistration]);
      Out_PosAsAttrs(nClusterPos); Out_XAttrEnd; Out_AdjectiveList(nConsequent);
      Out_Type(nClusterType); Out_XElEnd(ClusterRegistrationName[ExistentialRegistration]);
      end:
  ConditionalRegistration: with CClusterPtr(nContent)\uparrow do
      begin Out_XElStart(ClusterRegistrationName[ConditionalRegistration]);
      Out_PosAsAttrs(nClusterPos); Out_XAttrEnd; Out_AdjectiveList(nAntecedent);
      Out\_AdjectiveList(nConsequent); Out\_Type(nClusterType);
      Out_XElEnd(ClusterRegistrationName[ConditionalRegistration]);
      end:
  FunctorialRegistration: with FClusterPtr(nContent) \uparrow do
      begin Out_XElStart(ClusterRegistrationName[FunctorialRegistration]);
      Out_PosAsAttrs(nClusterPos); Out_XAttrEnd; Out_Term(nClusterTerm);
      Out_AdjectiveList(nConsequent);
      if nClusterType \neq nil then Out\_Type(nClusterType);
      Out_XElEnd(ClusterRegistrationName[FunctorialRegistration]);
      end:
  endcases;
See also section 1094.
This code is used in section 1080.
```

Mizar Parser

368 1094.

```
Identify-Registration-content =
(Pattern, Pattern,
  element LociEquality {
       attribute col { xsd:integer },
       attribute line { xsd:integer },
       Locus, Locus
    }*
  })
Sethood-Registration-content = (Type, Justification)
Reduction-Registration-content = (Term, Term)
\langle Emit XML for registration-related items (WSM) 1093\rangle +\equiv
itIdentify: with IdentifyRegistrationPtr(nContent) \uparrow do
    begin Out_Pattern(nOriginPattern); Out_Pattern(nNewPattern);
    if nEqLociList \neq nil then
      begin for i \leftarrow 0 to nEqLociList \uparrow.Count - 1 do
         with LociEqualityPtr(nEqLociList\uparrow.Items\uparrow[i])\uparrow do
           \textbf{begin} \ \ Out\_XElStart(XMLElemName[elLociEquality]); \ \ Out\_PosAsAttrs(nEqPos);
           Out_XAttrEnd; Out_Locus(nLeftLocus); Out_Locus(nRightLocus);
           Out_XElEnd(XMLElemName[elLociEquality]);
           end;
      end;
    end:
itPropertyRegistration: case PropertyRegistrationPtr(nContent)\uparrow.nPropertySort of
  sySethood: with SethoodRegistrationPtr(nContent) \uparrow do
      begin Out_Type(nSethoodType); Out_Justification(nJustification, nBlock);
      end:
  endcases;
itReduction: with ReduceRegistrationPtr(nContent) \uparrow do
    begin Out_Term(nOriginTerm); Out_Term(nNewTerm);
    end
```

```
1095.
       Emitting an item.
Item = element Item {
  attribute kind { text },
  Item-contents-attribute?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  attribute posline { xsd:integer },
  attribute poscol { xsd:integer },
  (Block | Item-contents)?
}
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
procedure OutWSMizFileObj.Out_Item(aWSItem: WSItemPtr);
  var i, j: integer;
  begin with aWSItem \uparrow do
    begin CurPos \leftarrow nItemPos; Out\_XElStart(XMLElemName[elItem]);
    Out_XAttr(XMLAttrName[atKind], ItemName[nItemKind]);
    if nContent \neq nil then Out\_ItemContentsAttr(aWsItem);
    Out_PosAsAttrs(nItemPos); Out_XIntAttr(XMLAttrName[atPosLine], nItemEndPos.Line);
    Out_XIntAttr(XMLAttrName[atPosCol], nItemEndPos.Col); Out_XAttrEnd;
    if nContent = nil then
      begin if nBlock \neq nil then Out\_Block(nBlock);
      end
    else Out_ItemContents(aWsItem);
    Out_XElEnd(XMLElemName[elItem]);
    end;
  end;
1096.
       Writing out to an XML file.
procedure Write_WSMizArticle(aWSTextProper: wsTextProperPtr; aFileName: string);
  var lWSMizOutput: OutWSMizFilePtr;
  begin InitScannerNames; \ lWSMizOutput \leftarrow new(OutWSMizFilePtr, OpenFile(aFileName));
  lWSMizOutput \uparrow .nMizarAppearance \leftarrow true; lWSMizOutput \uparrow .Out\_TextProper(aWSTextProper);
  dispose(lWSMizOutput, Done);
  end;
```

Section 21.12. READING WSM FILES (DEFERRED)

1097. Reading a WSM file amounts to reading an XML file, which means that the XMLInStream class (§528) is a natural parent class. Recall, the state of the XMLInStream contains the current start tag and a dictionary for the attributes and their values.

The code is a "mirror image" to writing XML files, and the XML schema guides the implementation. $\langle \text{Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv$ $In WSMizFilePtr = \uparrow In WSMizFileObj;$ $InWSMizFileObj = \mathbf{object} \ (XMLInStreamObj)$ nDisplayInformationOnScreen: boolean; constructor OpenFile(const aFileName: string); **destructor** Done: virtual: **function** GetAttrValue(**const** aAttrName: string): string; **function** GetAttrPos: Position: **function** Read_TextProper: wsTextProperPtr; virtual; **function** Read_Block: wsBlockPtr; virtual; **function** Read_Item: wsItemPtr; virtual; **procedure** Read_ItemContentsAttr(aItem: wsItemPtr; var aShape: string); virtual; **procedure** Read_ItemContents(aItem: wsItemPtr; const aShape: string); virtual; **function** Read_TermList: PList; virtual; **function** Read_Adjective: AdjectiveExpressionPtr; virtual; **function** Read_AdjectiveList: PList; virtual; **function** Read_Type: TypePtr; virtual; **function** Read_Variable: VariablePtr; virtual; **function** Read_ImplicitlyQualifiedSeqment: ImplicitlyQualifiedSeqmentPtr; virtual; **function** Read_VariableSegment: QualifiedSegmentPtr; virtual; **function** Read_PrivatePredicativeFormula: PrivatePredicativeFormulaPtr; virtual; **function** Read_Formula: FormulaPtr; virtual; **function** Read_SimpleTerm: SimpleTermPtr; virtual; **function** Read_PrivateFunctorTerm: PrivateFunctorTermPtr; virtual; **function** Read_InternalSelectorTerm: InternalSelectorTermPtr; virtual; **function** Read_Term: TermPtr; virtual; **function** Read_TypeList: PList; virtual; function Read_Locus: LocusPtr; virtual; **function** Read_Loci: PList; virtual; **function** Read_ModePattern: ModePatternPtr; virtual; **function** Read_AttributePattern: AttributePatternPtr; virtual; **function** Read_FunctorPattern: FunctorPatternPtr; virtual; **function** Read_PredicatePattern: PredicatePatternPtr; virtual; **function** Read_Pattern: PatternPtr; virtual; **function** Read_Definiens: DefiniensPtr; virtual; **function** Read_ReservationSegment: ReservationSegmentPtr; virtual; **function** Read_SchemeNameInSchemeHead: SchemePtr; virtual; function Read_Label: LabelPtr; virtual; **function** Read_Proposition: PropositionPtr; virtual; **function** Read_CompactStatement: CompactStatementPtr; virtual; $\mathbf{function}\ \textit{Read_LocalReference}\colon \textit{LocalReferencePtr};\ \textit{virtual};$ **function** Read_References: PList; virtual: **function** Read_StraightforwardJustification: StraightforwardJustificationPtr; virtual; **function** Read_SchemeJustification: SchemeJustificationPtr; virtual; **function** Read_Justification: JustificationPtr; virtual; function Read_RegularStatement(const aShape: string): RegularStatementPtr; virtual; end;

```
1098.
        Constructor.
\langle Implementation for wsmarticle.pas 854\rangle + \equiv
constructor InWSMizFileObj.OpenFile(const aFileName: string);
  begin inherited OpenFile(aFileName); nDisplayInformationOnScreen \leftarrow false;
  end:
destructor In WSMizFileObj.Done;
  begin inherited Done;
  end;
        Getting the value for an attribute. Returns nil if there is no attribute with the given name. (Recall
(\S490), an XMLAttr is just a wrapper around a string nValue.)
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function InWSMizFileObj.GetAttrValue(const aAttrName: string): string;
  var lObj: PObject;
  begin result \leftarrow ``; lObj \leftarrow nAttrVals.ObjectOf(aAttrName);
  if lObj \neq nil then result \leftarrow XMLAttrPtr(lObj)\uparrow.nValue;
  end;
1100. We can query for the position of the XML attribute.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function In WSMizFileObj.GetAttrPos: Position;
  var lLine, lCol: XMLAttrPtr; lCode: integer;
  begin result.Line \leftarrow 1; result.Col \leftarrow 1;
  lLine \leftarrow XMLAttrPtr(nAttrVals.ObjectOf(XMLAttrName[atLine]));
  lCol \leftarrow XMLAttrPtr(nAttrVals.ObjectOf(XMLAttrName[atCol]));
  if (lLine \neq nil) \land (lCol \neq nil) then
    begin Val(lLine \uparrow .nValue, result.Line, lCode); Val(lCol \uparrow .nValue, result.Col, lCode);
    end;
  end;
```

```
1101.
        The state of the WSM parser may be described with a handful of lookup tables.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
\mathbf{var}\ ElemLookupTable\ ,\ AttrLookupTable\ ,\ BlockLookUpTable\ ,\ ItemLookUpTable\ ,\ FormulaKindLookupTable\ ,
       TermKindLookupTable, PatternKindLookupTable, CorrectnessKindLookupTable,
      PropertyKindLookupTable: MSortedStrList;
procedure InitWSLookupTables;
  var e: XMLElemKind; a: XMLAttrKind; b: BlockKind; i: ItemKind; f: FormulaSort; t: TermSort;
    p: PropertyKind; c: CorrectnessKind;
  begin ElemLookupTable.Init(Ord(High(XMLElemKind)) + 1);
  AttrLookup Table.Init(Ord(High(XMLAttrKind)) + 1);
  BlockLookup Table.Init(Ord(High(BlockKind)) + 1); ItemLookup Table.Init(Ord(High(ItemKind)) + 1);
  FormulaKindLookupTable.Init(Ord(High(FormulaSort)) + 1);
  TermKindLookup Table . Init (Ord (High (TermSort)) + 1);
  PatternKindLookupTable.Init(Ord(itDefStruct) - Ord(itDefPred) + 1);
  CorrectnessKindLookupTable.Init(ord(High(CorrectnessKind)) + 1);
  PropertyKindLookupTable.Init(ord(High(PropertyKind)) + 1);
  for e \leftarrow Low(XMLElemKind) to High(XMLElemKind) do
    ElemLookup Table.Insert(new(MStrPtr,Init(XMLElemName[e])));
  for a \leftarrow Low(XMLAttrKind) to High(XMLAttrKind) do
    AttrLookup Table.Insert(new(MStrPtr, Init(XMLAttrName[a])));
  for b \leftarrow Low(BlockKind) to High(BlockKind) do
    BlockLookup Table.Insert(new(MStrPtr,Init(BlockName[b])));
  for i \leftarrow Low(ItemKind) to High(ItemKind) do
    ItemLookupTable.Insert(new(MStrPtr,Init(ItemName[i])));
  for f \leftarrow Low(FormulaSort) to High(FormulaSort) do
    Formula Kind Lookup Table . Insert(new(MStrPtr, Init(Formula Name[f])));
  for t \leftarrow Low(TermSort) to High(TermSort) do
    TermKindLookupTable.Insert(new(MStrPtr, Init(TermName[t])));
  for i \leftarrow itDefPred to itDefStruct do
    PatternKindLookupTable.Insert(new(MStrPtr,Init(DefPatternName[i])));
  for p \leftarrow Low(PropertyKind) to High(PropertyKind) do
    PropertyKindLookupTable.Insert(new(MStrPtr,Init(PropertyName[p])));
  for c \leftarrow Low(CorrectnessKind) to High(CorrectnessKind) do
    CorrectnessKindLookupTable.Insert(new(MStrPtr,Init(CorrectnessName[c])));
  end;
        We also need to free the memory consumed by the lookup tables.
1102.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure Dispose WSLookup Tables;
  begin ElemLookupTable.Done; AttrLookupTable.Done; BlockLookupTable.Done;
  ItemLookup Table . Done; Formula KindLookup Table . Done; TermKindLookup Table . Done;
  CorrectnessKindLookupTable.Done; PropertyKindLookupTable.Done;
  end;
```

1103. We can recall, from the XML dictionary module (§462), the different kinds of XML elements as specified by an enumerated constant. This converts the "nr" attribute to the human readable equivalents.

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv function Str2XMLElemKind(aStr:string): XMLElemKind; var lNr:integer; begin lNr \leftarrow ElemLookupTable.IndexOfStr(aStr); if lNr > -1 then Str2XMLElemKind \leftarrow XMLElemKind(lNr) else Str2XMLElemKind \leftarrow elUnknown; end;
```

1104. Like the previous function, this converts the "nr" attribute for a WSM Mizar attribute XML element into a human readable form.

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv function Str2XMLAttrKind(aStr:string): XMLAttrKind; var lNr:integer; begin lNr \leftarrow AttrLookupTable.IndexOfStr(aStr); if lNr > -1 then Str2XMLAttrKind \leftarrow XMLAttrKind(lNr) else Str2XMLAttrKind \leftarrow atUnknown; end;
```

1105. The "kinds" of different syntactic classes were introduced earlier in wsmarticle.pas, now we want to translate them into human readable form.

```
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
function Str2BlockKind(aStr: string): BlockKind;
  var lNr: integer;
  begin lNr \leftarrow BlockLookupTable.IndexOfStr(aStr);
  if lNr > -1 then Str2BlockKind \leftarrow BlockKind(lNr)
  else Str2BlockKind \leftarrow blMain;
  end:
function Str2ItemKind(aStr: string): ItemKind;
  var lNr: integer;
  begin lNr \leftarrow ItemLookupTable.IndexOfStr(aStr);
  if lNr > -1 then Str2ItemKind \leftarrow ItemKind(lNr)
  else Str2ItemKind \leftarrow itIncorrItem;
  end:
function Str2PatterenKind(aStr: string): ItemKind;
  var lNr: integer;
  begin lNr \leftarrow PatternKindLookupTable.IndexOfStr(aStr);
  if lNr > -1 then Str2PatterenKind \leftarrow ItemKind(Ord(ItDefPred) + lNr)
  else Str2PatterenKind \leftarrow itIncorrItem;
  end;
function Str2FormulaKind(aStr:string): FormulaSort;
  var lNr: integer;
  begin lNr \leftarrow FormulaKindLookupTable.IndexOfStr(aStr);
  if lNr > -1 then Str2FormulaKind \leftarrow FormulaSort(lNr)
  else Str2FormulaKind \leftarrow wsErrorFormula;
  end:
```

```
§1106
                                                                                                             375
         Mizar Parser
                                                                     READING WSM FILES (DEFERRED)
1106.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function Str2TermKind(aStr:string): TermSort;
  var lNr: integer;
  begin lNr \leftarrow TermKindLookupTable.IndexOfStr(aStr);
  if lNr > -1 then Str2TermKind \leftarrow TermSort(lNr)
  else Str2TermKind \leftarrow wsErrorTerm;
  end:
function Str2PropertyKind(aStr:string): PropertyKind;
  var lNr: integer;
  begin lNr \leftarrow PropertyKindLookupTable.IndexOfStr(aStr);
  if lNr > -1 then Str2PropertyKind \leftarrow PropertyKind(lNr)
  end:
function Str2CorrectnessKind(aStr:string): CorrectnessKind;
  var lNr: integer;
  begin lNr \leftarrow CorrectnessKindLookupTable.IndexOfStr(aStr);
  if lNr > -1 then Str2CorrectnessKind \leftarrow CorrectnessKind(lNr)
  end;
Subsection 21.12.1. Parsing types
1107.
        Reading a "term list" just iteratively invokes Read_Term (§1121) until all the children have been
read.
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
function In WSMizFileObj.Read_TermList: PList;
  begin result \leftarrow new(PList, Init(0)):
  while nState \neq eEnd do result \uparrow .Insert(Read\_Term);
  end:
1108. An adjective is either "positive" (i.e., not negated) or "negative" (i.e., negated). We handle the first
case in the "true" branch, and the second case in the "false" branch.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function In WSMizFileObj.Read_Adjective: AdjectiveExpressionPtr;
  var lAttrNr: integer; lPos: Position; lNoneOcc: Boolean;
  begin if nElName = AdjectiveSortName[wsAdjective] then
    begin lPos \leftarrow GetAttrPos; lAttrNr \leftarrow GetIntAttr(XMLAttrName[atNr]); NextElementState;
    result \leftarrow new(AdjectivePtr, Init(lPos, lAttrNr, Read\_TermList)); NextElementState;
  else begin lPos \leftarrow GetAttrPos; NextElementState;
    result \leftarrow new(NegatedAdjectivePtr, Init(lPos, Read\_Adjective)); NextElementState;
    end:
  end;
1109. Reading a list of adjectives just iterates over the children of an element.
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
function In WSMizFileObj.Read_AdjectiveList: PList;
  begin result \leftarrow new(Plist, Init(0)); NextElementState;
```

while $nState \neq eEnd$ do $result \uparrow .Insert(Read_Adjective);$

NextElementState;

end;

376 Parsing types Mizar Parser $\S1110$

1110. There are three valid Mizar types: "standard" types, structure types, and expandable modes (i.e., a cluster of adjectives stacked atop a type). If the XML element fails to match these three, then we should produce an "incorrect type".

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function In WSMizFileObj.Read_Type: TypePtr;
  var lList: Plist; lPos: Position; lModeSymbol: integer;
  begin if nElName = TypeName[wsStandardType] then
    begin lPos \leftarrow GetAttrPos; lModeSymbol \leftarrow GetIntAttr(XMLAttrName[atNr]); NextElementState;
    result \leftarrow new(StandardTypePtr, Init(lPos, lModeSymbol, Read\_TermList)); NextElementState;
    end
  else if nElName = TypeName[wsStructureType] then
       begin lPos \leftarrow GetAttrPos; lModeSymbol \leftarrow GetIntAttr(XMLAttrName[atNr]); NextElementState;
       result \leftarrow new(StructTypePtr, Init(lPos, lModeSymbol, Read\_TermList)); NextElementState;
       end
    else if nElName = TypeName[wsClusteredType] then
         begin lPos \leftarrow GetAttrPos; NextElementState; lList \leftarrow Read\_AdjectiveList;
         result \leftarrow new(ClusteredTypePtr, Init(lPos, lList, Read\_Type)); NextElementState;
         end
       else begin lPos \leftarrow GetAttrPos; NextElementState; result \leftarrow new(IncorrectTypePtr, Init(lPos));
         NextElementState;
         end
  end;
Subsection 21.12.2. Parsing formulas
1111. Parsing a variable from XML just requires reading the attributes, since it is an empty-element.
\langle Implementation for wsmarticle.pas 854\rangle +=
function In WSMizFileObj.Read_Variable: VariablePtr;
  var lPos: Position; lNr: integer;
  begin lPos \leftarrow GetAttrPos; lNr \leftarrow GetIntAttr(XMLAttrName[atIdNr]);
  NextElementState; { closes the variable's tag }
  result \leftarrow new(VariablePtr, Init(lPos, lNr));
  NextElementState; { starts the next tag }
  end;
1112. Implicitly qualified variables are just wrappers around a variable.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function In WSMizFileObj.Read_ImplicitlyQualifiedSegment: ImplicitlyQualifiedSegmentPtr;
  var lPos: Position;
  begin lPos \leftarrow GetAttrPos; NextElementState;
```

 $result \leftarrow new(ImplicitlyQualifiedSegmentPtr, Init(lPos, Read_Variable)); NextElementState;$

end:

§1113 Mizar Parser PARSING FORMULAS 377

Recall (§1025) that a "qualified segment" is either implicit (i.e., a wrapper around a single variable) or explicit (i.e., an element whose children are variables and a type). \langle Implementation for wsmarticle.pas $854 \rangle + \equiv$ **function** In WSMizFileObj.Read_VariableSegment: QualifiedSegmentPtr; var lPos: Position; lVar: VariablePtr; lList: PList; **begin** if nElName = SegmentKindName[ikImplQualifiedSegm] then **begin** $result \leftarrow Read_ImplicitlyQualifiedSegment;$ else if nElName = SegmentKindName[ikExplQualifiedSegm] then **begin** $lPos \leftarrow GetAttrPos; NextElementState; lList \leftarrow new(PList, Init(0));$ NextElementState; { read the variables } while $(nState = eStart) \land (nElName = XMLElemName[elVariable])$ do $lList \uparrow .Insert(Read_Variable);$ NextElementState; { read the type } $result \leftarrow new(ExplicitlyQualifiedSegmentPtr, Init(lPos, lList, Read_Type));$ NextElementState; { start the next tag } end end; 1114. Private predicates are empty elements, so we only need to read their attributes. \langle Implementation for wsmarticle.pas $854 \rangle + \equiv$ **function** In WSMizFileObj.Read_PrivatePredicativeFormula: PrivatePredicativeFormulaPtr; **var** lPos: Position; lNr: integer; **begin** $lPos \leftarrow GetAttrPos$; $lNr \leftarrow GetIntAttr(XMLAttrName[atIdNr])$; NextElementState; $Result \leftarrow new(PrivatePredicativeFormulaPtr, Init(lPos, lNr, Read_TermList)); NextElementState;$ end;

378 Parsing formulas Mizar Parser $\S1115$

```
1115.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function In WSMizFileObj.Read_Formula: FormulaPtr;
  var lPos: Position; lNr: integer; lList: PList; lFrm: FormulaPtr; lTrm: TermPtr;
    lSqm: QualifiedSegmentPtr;
  begin case Str2FormulaKind(nElName) of
  wsNegatedFormula: begin lPos \leftarrow GetAttrPos; NextElementState;
    result \leftarrow new(NegativeFormulaPtr, Init(lPos, Read\_Formula)); NextElementState;
    end;
  ⟨ Parse XML for formula with binary connective 1116⟩;
  wsFlexaryConjunctiveFormula: begin lPos \leftarrow GetAttrPos; NextElementState; lFrm \leftarrow Read\_Formula;
    result \leftarrow new(FlexaryConjunctiveFormulaPtr, Init(lPos, lFrm, Read\_Formula)); NextElementState;
  wsFlexaryDisjunctiveFormula: begin lPos \leftarrow GetAttrPos; NextElementState; lFrm \leftarrow Read\_Formula;
    result \leftarrow new(FlexaryDisjunctiveFormulaPtr, Init(lPos, lFrm, Read\_Formula)); NextElementState;
  ⟨ Parse XML for predicate-based formula 1117⟩;
  wsAttributiveFormula: \mathbf{begin} \ lPos \leftarrow GetAttrPos; \ NextElementState; \ lTrm \leftarrow Read\_Term;
    Result \leftarrow new(AttributiveFormulaPtr, Init(lPos, lTrm, Read\_AdjectiveList)); NextElementState;
    end;
  wsQualifyinqFormula: begin lPos \leftarrow GetAttrPos; NextElementState; lTrm \leftarrow Read\_Term;
    Result \leftarrow new(QualifyingFormulaPtr, Init(lPos, lTrm, Read\_Type)); NextElementState;
    end:
  wsUniversalFormula: \mathbf{begin} \ lPos \leftarrow GetAttrPos: \ NextElementState: \ lSqm \leftarrow Read\_VariableSegment:
    Result \leftarrow new(UniversalFormulaPtr, Init(lPos, lSgm, Read\_Formula)); NextElementState;
  wsExistentialFormula: begin \ lPos \leftarrow GetAttrPos; \ NextElementState; \ lSgm \leftarrow Read\_VariableSegment;
    Result \leftarrow new(ExistentialFormulaPtr, Init(lPos, lSgm, Read\_Formula)); NextElementState;
    end:
  wsContradiction: begin \ lPos \leftarrow GetAttrPos; \ NextElementState;
    result \leftarrow new(ContradictionFormulaPtr, Init(lPos)); NextElementState;
    end:
  wsThesis: \mathbf{begin} \ lPos \leftarrow GetAttrPos; \ NextElementState; \ result \leftarrow new(ThesisFormulaPtr, Init(lPos));
    NextElementState;
    end;
  wsErrorFormula: begin \ lPos \leftarrow GetAttrPos; \ NextElementState;
    result \leftarrow new(IncorrectFormulaPtr, Init(lPos)); NextElementState;
    end:
  endcases:
  end;
```

§1116 Mizar Parser PARSING FORMULAS 379

1116. For formulas with binary connectives, we read both arguments. \langle Parse XML for formula with binary connective 1116 $\rangle \equiv$ wsConjunctiveFormula: begin $lPos \leftarrow GetAttrPos$; NextElementState; $lFrm \leftarrow Read_Formula$; $result \leftarrow new(ConjunctiveFormulaPtr, Init(lPos, lFrm, Read_Formula)); NextElementState;$ end: wsDisjunctiveFormula: begin $lPos \leftarrow GetAttrPos$; NextElementState; $lFrm \leftarrow Read_Formula$; $result \leftarrow new(DisjunctiveFormulaPtr, Init(lPos, lFrm, Read_Formula)); NextElementState;$ wsConditionalFormula: begin $lPos \leftarrow GetAttrPos$; NextElementState; $lFrm \leftarrow Read_Formula$; $result \leftarrow new(ConditionalFormulaPtr, Init(lPos, lFrm, Read_Formula)); NextElementState;$ end: wsBiconditionalFormula: begin $lPos \leftarrow GetAttrPos$; NextElementState; $lFrm \leftarrow Read_Formula$; $result \leftarrow new(BiconditionalFormulaPtr, Init(lPos, lFrm, Read_Formula)); NextElementState;$ end This code is used in section 1115. 1117. $\langle \text{ Parse XML for predicate-based formula } 1117 \rangle \equiv$ $wsPredicativeFormula: begin lPos \leftarrow GetAttrPos; lNr \leftarrow GetIntAttr(XMLAttrName[atNr]);$ NextElementState; NextElementState; { Arguments } $lList \leftarrow Read_TermList; NextElementState; { Arguments }$ NextElementState; { Arguments } $Result \leftarrow new(PredicativeFormulaPtr, Init(lPos, lNr, lList, Read_TermList)); NextElementState;$ NextElementState;end: $wsRightSideOfPredicativeFormula: begin lPos \leftarrow GetAttrPos; lNr \leftarrow GetIntAttr(XMLAttrName[atNr]);$ NextElementState; NextElementState; { Arguments } $Result \leftarrow new(RightSideOfPredicativeFormulaPtr, Init(lPos, lNr, Read_TermList)); NextElementState;$ NextElementState;end: $wsMultiPredicativeFormula: begin lPos \leftarrow GetAttrPos; NextElementState; lList \leftarrow new(PList, Init(0));$ while $nState \neq eEnd$ do $lList \uparrow .Insert(Read_Formula)$; $result \leftarrow new(MultiPredicativeFormulaPtr, Init(lPos, lList)); NextElementState;$ wsPrivatePredicateFormula: **begin** $Result \leftarrow Read_PrivatePredicativeFormula$;

end

This code is used in section 1115.

380 PARSING TERMS Mizar Parser $\S1118$

Subsection 21.12.3. Parsing terms

```
1118.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function In WSMizFileObj.Read_SimpleTerm: SimpleTermPtr;
  var lPos: Position; lNr: integer;
  begin lPos \leftarrow GetAttrPos; lNr \leftarrow GetIntAttr(XMLAttrName[atIdNr]); NextElementState;
  result \leftarrow new(SimpleTermPtr, Init(lPos, lNr)); \ NextElementState;
  end;
1119.
\langle Implementation for wsmarticle.pas 854\rangle + \equiv
\textbf{function} \ \textit{InWSMizFileObj}. \textit{Read\_PrivateFunctorTerm}: \ \textit{PrivateFunctorTermPtr};
  var lPos: Position; lNr: integer;
  begin lPos \leftarrow GetAttrPos; lNr \leftarrow GetIntAttr(XMLAttrName[atIdNr]); NextElementState;
  result \leftarrow new(PrivateFunctorTermPtr, Init(lPos, lNr, Read\_TermList)); NextElementState;
  end;
1120.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
{\bf function}\ In WSMizFile Obj. Read\_Internal Selector Term:\ Internal Selector TermPtr;
  var lPos: Position; lNr: integer;
  begin lPos \leftarrow GetAttrPos; lNr \leftarrow GetIntAttr(XMLAttrName[atNr]); NextElementState;
  result \leftarrow new(InternalSelectorTermPtr, Init(lPos, lNr)); NextElementState;
  end;
```

§1121 Mizar Parser PARSING TERMS 381

1121.

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function In WSMizFileObj.Read_Term: TermPtr;
     var lPos, lRPos: Position; lNr, lRNr: integer; lList: PList; lTrm: TermPtr;
     begin case Str2TermKind(nElName) of
     wsPlaceholderTerm: begin lPos \leftarrow GetAttrPos; lNr \leftarrow GetIntAttr(XMLAttrName[atNr]);
           NextElementState; result \leftarrow new(PlaceholderTermPtr, Init(lPos, lNr)); NextElementState;
           end:
     wsSimpleTerm: \mathbf{begin} \ result \leftarrow Read\_SimpleTerm;
           end:
     wsNumeralTerm: begin lPos \leftarrow GetAttrPos: lNr \leftarrow GetIntAttr(XMLAttrName[atNumber]):
           NextElementState; result \leftarrow new(NumeralTermPtr, Init(lPos, lNr)); NextElementState;
     wsInfixTerm: begin lPos \leftarrow GetAttrPos; lNr \leftarrow GetIntAttr(XMLAttrName[atNr]); NextElementState;
           NextElementState; { Arguments }
           lList \leftarrow Read\_TermList; NextElementState; { Arguments }
           NextElementState; { Arguments }
           result \leftarrow new(InfixTermPtr, Init(lPos, lNr, lList, Read\_TermList)); NextElementState;
           NextElementState;
           end:
     wsCircumfixTerm: \mathbf{begin} \ lPos \leftarrow GetAttrPos; \ lNr \leftarrow GetIntAttr(XMLAttrName[atNr]);
           NextElementState; NextElementState; lRNr \leftarrow GetIntAttr(XMLAttrName[atNr]);
           lRPos \leftarrow GetAttrPos; NextElementState;
           result \leftarrow new(CircumfixTermPtr, Init(lPos, lNr, lRNr, Read\_TermList)); NextElementState;
           end:
     wsPrivateFunctorTerm: begin result \leftarrow Read\_PrivateFunctorTerm;
           end;
     wsAqqreqateTerm: \mathbf{begin} \ lPos \leftarrow GetAttrPos; \ lNr \leftarrow GetIntAttr(XMLAttrName[atNr]);
           NextElementState; result \leftarrow new(AggregateTermPtr, Init(lPos, lNr, Read\_TermList));
           NextElementState;
           end:
     wsSelectorTerm: begin lPos \leftarrow GetAttrPos; lNr \leftarrow GetIntAttr(XMLAttrName[atNr]);
           NextElementState; result \leftarrow new(SelectorTermPtr, Init(lPos, lNr, Read\_Term)); NextElementState;
     wsInternalSelectorTerm: result \leftarrow Read\_InternalSelectorTerm;
     wsForgetfulFunctorTerm: \mathbf{begin} \ lPos \leftarrow GetAttrPos; \ lNr \leftarrow GetIntAttr(XMLAttrName[atNr]);
           NextElementState; result \leftarrow new(ForgetfulFunctorTermPtr, Init(lPos, lNr, Read\_Term));
           NextElementState;
           end;
     wsInternalForgetfulFunctorTerm: begin lPos \leftarrow GetAttrPos; lNr \leftarrow GetIntAttr(XMLAttrName[atNr]);
           NextElementState; result \leftarrow new(InternalForgetfulFunctorTermPtr, Init(lPos, lNr));
           NextElementState;
     wsFraenkelTerm: begin lPos \leftarrow GetAttrPos; NextElementState; lList \leftarrow new(PList, Init(0));
           while (nState = eStart) \land ((nElName = SeqmentKindName[ikImplQualifiedSeqm]) \lor (nElName = SeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqme
                            SegmentKindName[ikExplQualifiedSegm])) do lList\uparrow.Insert(Read\_VariableSegment);
           lTrm \leftarrow Read\_Term; result \leftarrow new(FraenkelTermPtr, Init(lPos, lList, lTrm, Read\_Formula));
           NextElementState;
           end;
     wsSimpleFraenkelTerm: begin lPos \leftarrow GetAttrPos; NextElementState; lList \leftarrow new(PList, Init(0));
           while (nState = eStart) \land ((nElName = SegmentKindName[ikImplQualifiedSegm]) \lor (nElName = SegmentKindName[ikImplQualifiedSegmentKindName]) \lor (nElName = Segme
                            SegmentKindName[ikExplQualifiedSegm])) do lList\uparrow.Insert(Read\_VariableSegment);
```

382 PARSING TERMS Mizar Parser $\S1121$

```
lTrm \leftarrow Read\_Term; result \leftarrow new(SimpleFraenkelTermPtr, Init(lPos, lList, lTrm));
     NextElementState;
  wsQualificationTerm: begin lPos \leftarrow GetAttrPos; NextElementState; lTrm \leftarrow Read\_Term;
     Result \leftarrow new(QualifiedTermPtr, Init(lPos, lTrm, Read\_Type)); NextElementState;
  wsExactlyTerm: begin lPos \leftarrow GetAttrPos; NextElementState:
     Result \leftarrow new(ExactlyTermPtr, Init(lPos, Read\_Term)); NextElementState;
     end:
  wsGlobalChoiceTerm: begin lPos \leftarrow GetAttrPos; NextElementState;
     Result \leftarrow new(ChoiceTermPtr, Init(lPos, Read\_Type)); NextElementState;
  wsItTerm: begin lPos \leftarrow GetAttrPos; NextElementState; Result \leftarrow new(ItTermPtr, Init(lPos));
     NextElementState:
     end;
  wsErrorTerm: begin \ lPos \leftarrow GetAttrPos; \ NextElementState;
     Result \leftarrow new(IncorrectTermPtr, Init(lPos)); NextElementState;
  endcases;
  end;
Subsection 21.12.4. Parsing text items
1122.
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
function In WSMizFileObj.Read_TypeList: PList;
  begin NextElementState; result \leftarrow new(PList, Init(0));
  while nState \neq eEnd do result \uparrow .Insert(Read\_Type);
  NextElementState;
  end:
1123.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function In WSMizFileObj.Read_Locus: LocusPtr;
  var lPos: Position; lNr: integer;
  begin lPos \leftarrow GetAttrPos; lNr \leftarrow GetIntAttr(XMLAttrName[atIdNr]); NextElementState;
  result \leftarrow new(LocusPtr, Init(lPos, lNr)); NextElementState;
  end:
1124.
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
function In WSMizFileObj.Read_Loci: PList;
  begin NextElementState; result \leftarrow new(PList, Init(0));
  while nState \neq eEnd do result \uparrow .Insert(Read\_Locus);
  NextElementState;
  end;
```

 $\S1125$ Mizar Parser PARSING TEXT ITEMS 383

```
1125.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function In WSMizFileObj.Read_ModePattern: ModePatternPtr;
  var lPos: Position; lNr: integer;
  begin lPos \leftarrow GetAttrPos; lNr \leftarrow GetIntAttr(XMLAttrName[atNr]); NextElementState;
  result \leftarrow new(ModePatternPtr, Init(lPos, lNr, Read\_Loci)); NextElementState;
  end;
1126.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
\textbf{function} \ \textit{InWSMizFileObj}. \textit{Read\_AttributePattern}: \ \textit{AttributePatternPtr};
  var lPos: Position; lNr: integer; lArg: LocusPtr;
  begin lPos \leftarrow GetAttrPos; lNr \leftarrow GetIntAttr(XMLAttrName[atNr]); NextElementState;
  lArg \leftarrow Read\_Locus; result \leftarrow new(AttributePatternPtr, Init(lPos, lArg, lNr, Read\_Loci));
  NextElementState;
  end;
1127.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function In WSMizFileObj.Read_FunctorPattern: FunctorPatternPtr;
  var lPos, lRPos: Position; lNr, lRNr: integer; lArgs: PList;
  begin if nState = eStart then
     if nElName = FunctorPatternName [InfixFunctor] then
       \textbf{begin} \ lPos \leftarrow GetAttrPos; \ lNr \leftarrow GetIntAttr(XMLAttrName[atNr]); \ NextElementState;
       lArgs \leftarrow Read\_Loci; result \leftarrow new(InfixFunctorPatternPtr, Init(lPos, lArgs, lNr, Read\_Loci));
       NextElementState;
       end
     else if nElName = FunctorPatternName[CircumfixFunctor] then
         begin lPos \leftarrow GetAttrPos; lNr \leftarrow GetIntAttr(XMLAttrName[atNr]); NextElementState;
         lRNr \leftarrow GetIntAttr(XMLAttrName[atNr]); NextElementState; NextElementState;
         result \leftarrow new(CircumfixFunctorPatternPtr, Init(lPos, lNr, lRNr, Read\_Loci)); NextElementState;
          end;
  end;
1128.
\langle Implementation for wsmarticle.pas 854\rangle + \equiv
function In WSMizFileObj.Read_PredicatePattern: PredicatePatternPtr;
  var lPos, lRPos: Position; lNr, lRNr: integer; lArgs: PList;
  begin lPos \leftarrow GetAttrPos; lNr \leftarrow GetIntAttr(XMLAttrName[atNr]); NextElementState;
  lArgs \leftarrow Read\_Loci; result \leftarrow new(PredicatePatternPtr, Init(lPos, lArgs, lNr, Read\_Loci));
  NextElementState;
  end:
```

384 PARSING TEXT ITEMS Mizar Parser $\S1129$

1129.

```
 \langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv \\ \text{function } InWSMizFileObj.Read\_Pattern: PatternPtr; \\ \text{begin case } Str2PatterenKind(nElName) \text{ of} \\ itDefPred: result \leftarrow Read\_PredicatePattern; \\ itDefFunc: result \leftarrow Read\_FunctorPattern; \\ itDefMode: result \leftarrow Read\_ModePattern; \\ itDefAttr: result \leftarrow Read\_AttributePattern; \\ \text{othercases if } (nElName = FunctorPatternName[InfixFunctor]) \lor (nElName = FunctorPatternName[CircumfixFunctor]) \text{ then } result \leftarrow Read\_FunctorPattern \\ \text{else } result \leftarrow \text{nil}; \\ \text{endcases}; \\ \text{end}; \\ \end{cases}
```

 $\{1130$ Mizar Parser PARSING TEXT ITEMS 385

```
1130.
```

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function In WSMizFileObj.Read_Definiens: DefiniensPtr;
  var lPos: Position; lKind, lShape: string; lLab: LabelPtr; lExpr: PObject; lExpKind: ExpKind;
    lList: PList; lOtherwise: DefExpressionPtr;
  begin result \leftarrow nil;
  if (nState = eStart) \land (nElName = XMLElemName[elDefiniens]) then
    begin lPos \leftarrow GetAttrPos; lKind \leftarrow GetAttr(XMLAttrName[atKind]);
    lShape \leftarrow GetAttr(XMLAttrName[atShape]); NextElementState; lLab \leftarrow Read\_Label;
    if lKind = DefiniensKindName[SimpleDefiniens] then
       begin lExpKind \leftarrow exFormula;
       if lShape = ExpName[exTerm] then lExpKind \leftarrow exTerm;
       case lExpKind of
       exTerm: lExpr \leftarrow Read\_Term;
       exFormula: lExpr \leftarrow Read\_Formula;
       endcases:
       result \leftarrow new(SimpleDefiniensPtr, Init(lPos, lLab, new(DefExpressionPtr, Init(lExpKind, lExpr))));
       end
    else begin lList \leftarrow new(Plist, Init(0));
       while (nState = eStart) \land (nElName = XMLElemName[elPartialDefiniens]) do
         begin NextElementState; lExpKind \leftarrow exFormula;
         if lShape = ExpName[exTerm] then lExpKind \leftarrow exTerm;
         case lExpKind of
         exTerm: lExpr \leftarrow Read\_Term;
         exFormula: lExpr \leftarrow Read\_Formula;
         endcases; lList \uparrow .Insert(new(PartDefPtr, Init(new(DefExpressionPtr, Init(lExpKind, lExpr)),
              Read_Formula))); NextElementState;
         end;
       lOtherwise \leftarrow \mathbf{nil};
       if nState \neq eEnd then
         begin lExpKind \leftarrow exFormula;
         if lShape = ExpName[exTerm] then lExpKind \leftarrow exTerm;
         case lExpKind of
         exTerm: lExpr \leftarrow Read\_Term;
         exFormula: lExpr \leftarrow Read\_Formula;
         endcases; lOtherwise \leftarrow new(DefExpressionPtr, Init(lExpKind, lExpr));
         end:
       result \leftarrow new(ConditionalDefiniensPtr, Init(lPos, lLab, lList, lOtherwise))
       end:
    NextElementState;
    end;
  end;
```

386 Parsing text items Mizar Parser $\S1131$

```
1131.
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
function In WSMizFileObj.Read_Label: LabelPtr;
  var lLabPos: Position; lLabId: Integer;
  begin result \leftarrow nil;
  if (nState = eStart) \land (nElName = XMLElemName[elLabel]) then
    begin lLabId \leftarrow GetIntAttr(XMLAttrName[atIdNr]); \ lLabPos \leftarrow GetAttrPos; \ NextElementState;
    NextElementState; result \leftarrow new(LabelPtr, Init(lLabId, lLabPos));
    end;
  end;
1132.
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
function In WSMizFileObj.Read_Proposition: PropositionPtr;
  var lPos: Position; lLab: LabelPtr;
  begin NextElementState; lLab \leftarrow Read\_label;
  result \leftarrow new(PropositionPtr, Init(lLab, Read\_Formula, lPos)); NextElementState;
  end;
1133.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function In WSMizFileObj.Read_LocalReference: LocalReferencePtr;
  var lPos: Position; lNr: integer;
  begin lPos \leftarrow GetAttrPos; lNr \leftarrow GetIntAttr(XMLAttrName[atIdNr]); NextElementState;
  NextElementState; result \leftarrow new(LocalReferencePtr, Init(lNr, lPos));
  end;
1134.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function In WSMizFileObj.Read_References: PList;
  var lPos: Position; lNr, lFileNr: integer;
  begin result \leftarrow new(Plist, Init(0));
  while nState \neq eEnd do
    if nElName = ReferenceKindName[LocalReference] then
       begin result \( \). Insert (Read_LocalReference )
       end
    else if nElName = ReferenceKindName[TheoremReference] then
         begin lPos \leftarrow GetAttrPos; \ lFileNr \leftarrow GetIntAttr(XMLAttrName[atNr]);
         lNr \leftarrow GetIntAttr(XMLAttrName[atNumber]); NextElementState; NextElementState;
         result \uparrow . Insert(new(TheoremReferencePtr, Init(lFileNr, lNr, lPos)))
         end
       else if nElName = ReferenceKindName[DefinitionReference] then
            begin lPos \leftarrow GetAttrPos; lFileNr \leftarrow GetIntAttr(XMLAttrName[atNr]);
            lNr \leftarrow GetIntAttr(XMLAttrName[atNumber]); NextElementState; NextElementState;
            result \uparrow . Insert(new(DefinitionReferencePtr, Init(lFileNr, lNr, lPos)))
            end;
  end:
```

 $\S1135$ Mizar Parser PARSING TEXT ITEMS 387

```
1135.
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
function In WSMizFileObj.Read_ReservationSegment: ReservationSegmentPtr;
  var lList: PList;
  begin lList \leftarrow new(PList, Init(0)); NextElementState; {elVariables}
  while (nState = eStart) \land (nElName = XMLElemName[elVariable]) do lList \uparrow .Insert(Read\_Variable);
  NextElementState; result \leftarrow new(ReservationSegmentPtr, Init(lList, Read\_Type));
  end;
1136.
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
function In WSMizFileObj.Read_SchemeNameInSchemeHead: SchemePtr;
  var lNr: Integer; lPos: Position;
  begin lPos \leftarrow GetAttrPos; lNr \leftarrow GetIntAttr(XMLAttrName[atIdNr]);
  result \leftarrow new(SchemePtr, Init(lNr, lPos, nil, nil, nil));
  end;
1137.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function In WSMizFileObj.Read_CompactStatement: CompactStatementPtr;
  var lProp: PropositionPtr;
  begin lProp \leftarrow Read\_Proposition; result \leftarrow new(CompactStatementPtr, Init(lProp, Read\_Justification));
  end:
1138.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function In WSMizFileObj.Read_StraightforwardJustification: StraightforwardJustificationPtr;
  var lPos, lLinkPos: Position; lLinked: boolean;
  begin lPos \leftarrow GetAttrPos; NextElementState; lLinked \leftarrow false; lLinkPos \leftarrow lPos;
  if nelName = XMLElemName[elLink] then
     begin lLinked \leftarrow true; lLinkPos \leftarrow GetAttrPos; NextElementState; NextElementState;
  result \leftarrow new(StraightforwardJustificationPtr, Init(lPos, lLinked, lLinkPos));
  StraightforwardJustificationPtr(result)\uparrow.nReferences \leftarrow Read\_References; NextElementState;
  end;
1139.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function In WSMizFileObj.Read_SchemeJustification: SchemeJustificationPtr;
  var lInfPos, lPos: Position; lNr, lIdNr: integer;
  begin UnfPos \leftarrow GetAttrPos; Unr \leftarrow GetIntAttr(XMLAttrName[atNr]);
  lIdNr \leftarrow GetIntAttr(XMLAttrName[atIdNr]); lPos.Line \leftarrow GetIntAttr(XMLAttrName[atPosLine]);
  lPos.Col \leftarrow GetIntAttr(XMLAttrName[atPosCol]); NextElementState;
  result \leftarrow new(SchemeJustificationPtr, Init(UnfPos, lNr, lIdNr));
  SchemeJustificationPtr(result) \uparrow .nSchemeInfPos \leftarrow lPos;
  SchemeJustificationPtr(result) \uparrow .nReferences \leftarrow Read\_References; NextElementState;
  end;
```

388 PARSING TEXT ITEMS Mizar Parser $\S1140$

```
1140.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function In WSMizFileObj.Read_Justification: JustificationPtr;
  var lPos: Position;
  begin if nState = eStart then
    if nElName = InferenceName[infStraightforwardJustification] then
       result \leftarrow Read\_StraightforwardJustification
    else if nElName = InferenceName[infSchemeJustification] then result \leftarrow Read\_SchemeJustification
       else if nElName = InferenceName[infError] then
            begin lPos \leftarrow GetAttrPos; NextElementState;
            result \leftarrow new(JustificationPtr, Init(infError, lPos)); NextElementState;
            end
         else if nElName = InferenceName[infSkippedProof] then
              begin lPos \leftarrow GetAttrPos; NextElementState;
              result \leftarrow new(JustificationPtr, Init(infSkippedProof, lPos)); NextElementState;
            else result \leftarrow new(JustificationPtr, Init(infProof, CurPos));
  end;
1141.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function InWSMizFileObj.Read_RegularStatement(const aShape: string): RegularStatementPtr;
  var lPos: Position; lIdNr: integer; lTrm: TermPtr; lCStm: CompactStatementPtr; lLab: LabelPtr;
  begin if aShape = RegularStatementName[stDiffuseStatement] then
    begin lLab \leftarrow Read\_Label; result \leftarrow new(DiffuseStatementPtr, Init(lLab, stDiffuseStatement));
    end
  else if aShape = RegularStatementName[stCompactStatement] then
       begin result \leftarrow Read\_CompactStatement;
    else if aShape = RegularStatementName[stIterativeEquality] then
         begin lCStm \leftarrow Read\_CompactStatement; result \leftarrow new(IterativeEqualityPtr,
              Init(lCStm\uparrow.nProp, lCStm\uparrow.nJustification, new(PList, Init(0))));
         while (nState = eStart) \land (nElName = XMLElemName[elIterativeStep]) do
            begin lPos \leftarrow GetAttrPos; NextElementState; <math>lTrm \leftarrow Read\_Term;
            Iterative Equality Ptr(result) \uparrow . nIterSteps \uparrow . Insert(new(Iterative Step Ptr, Init(IPos, ITrm, Iterative Steps )) 
                 Read_Justification))); NextElementState;
            end;
         end;
  end;
```

 $\S1142$ Mizar Parser PARSING TEXT ITEMS 389

1142.

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure InWSMizFileObj.Read_ItemContentsAttr(aItem:wsItemPtr; var aShape:string);
  begin aShape \leftarrow ::;
  case aItem↑.nItemKind of
  itIncorrItem: ;
  itDefinition, itSchemeBlock, itSchemeHead, itTheorem, itAxiom, itReservation:;
  itSection:;
  itConclusion, itRegularStatement: aShape \leftarrow GetAttr(XMLAttrName[atShape]);
  itChoice, itReconsider, itPrivFuncDefinition, itPrivPredDefinition, itConstantDefinition, itGeneralization,
         itLociDeclaration, itExistentialAssumption, itExemplification, itPerCases, itCaseBlock;
  itCaseHead, itSupposeHead, itAssumption:;
  itCorrCond: aItem \uparrow .nContent \leftarrow new(CorrectnessConditionPtr, Init(CurPos,
         Str2CorrectnessKind(GetAttr(XMLAttrName[atCondition])), nil));
  itCorrectness: aItem \uparrow. nContent \leftarrow new(CorrectnessConditionsPtr, Init(CurPos, [], nil));
  itProperty: aShape \leftarrow GetAttr(XMLAttrName[atProperty]);
  itDefFunc: aShape \leftarrow GetAttr(XMLAttrName[atShape]);
  itDefPred, itDefMode, itDefAttr, itDefStruct, itPredSynonym, itPredAntonym, itFuncNotation,
         itModeNotation, itAttrSynonym, itAttrAntonym, itCluster, itIdentify, itReduction:;
  itPropertyRegistration: aShape \leftarrow GetAttr(XMLAttrName[atProperty]);
  itPragma: aItem \uparrow. nContent \leftarrow new(PragmaPtr, Init(XMLToStr(GetAttr(XMLAttrName[atSpelling]))));
  endcases;
  end;
```

390 Parsing text items Mizar Parser $\S1143$

1143.

```
\langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
procedure In WSMizFileObj.Read_ItemContents(aItem: wsItemPtr;
          const aShape: string);
     var lList, lCons, lConds, lVars, lFields, lTyps, lSels: PList; lType: TypePtr; lNr: Integer;
          lVar: VariablePtr; lLocus: LocusPtr; lTrm: TermPtr; lPos, lFieldSqmPos: Position;
          lRedefinition: boolean; lPattern: PatternPtr; lDef: HowToDefine; lPropertySort: PropertyKind;
     begin lPos \leftarrow CurPos;
     case aItem \uparrow .nItemKind of
     itIncorrItem:;
     itDefinition:;
     itSchemeBlock:;
     itSchemeHead: begin aItem \uparrow .nContent \leftarrow Read\_SchemeNameInSchemeHead; NextElementState;
          NextElementState; NextElementState; { elSchematicVariables }
          lList \leftarrow new(PList, Init(0));
          while (nState = eStart) \land ((nElName = SchemeSegmentName[PredicateSegment]) \lor (nElName = Scheme
                         SchemeSegmentName[FunctorSegment])) do
              if nElName = SchemeSegmentName[PredicateSegment] then
                    begin lPos \leftarrow GetAttrPos; NextElementState; lVars \leftarrow new(PList, Init(0)); NextElementState;
                              { elVariables }
                   while (nState = eStart) \land (nElName = XMLElemName[elVariable]) do
                         lVars \uparrow . Insert(Read\_Variable);
                    NextElementState;
                    lList \uparrow. Insert(new(PredicateSegmentPtr, Init(lPos, PredicateSegment, lVars, Read\_TypeList)));
                    NextElementState;
                    end
               else begin lPos \leftarrow GetAttrPos; NextElementState; lVars \leftarrow new(PList, Init(0));
                    NextElementState; { elVariables }
                    while (nState = eStart) \land (nElName = XMLElemName[elVariable]) do
                         lVars \uparrow .Insert(Read\_Variable);
                    NextElementState; \ lTyps \leftarrow Read\_TypeList; \ NextElementState;
                    lList\uparrow.Insert(new(FunctorSegmentPtr,Init(lPos,lVars,lTyps,Read\_Type)));\ NextElementState;
                    NextElementState;
                    end;
          SchemePtr(aItem\uparrow.nContent)\uparrow.nSchemeParams \leftarrow lList; NextElementState;
                    { elSchematicVariables }
          SchemePtr(aItem\uparrow.nContent)\uparrow.nSchemeConclusion \leftarrow Read\_Formula;\ lConds \leftarrow new(PList, Init(0));
          if (nState = eStart) \land (nElName = XMLElemName[elProvisionalFormulas]) then
              begin NextElementState;
               while (nState = eStart) \land (nElName = XMLElemName[elProposition]) do
                    lConds \uparrow . Insert(Read\_Proposition);
               NextElementState;
          SchemePtr(aItem\uparrow.nContent)\uparrow.nSchemePremises \leftarrow lConds;
     itTheorem: aItem \uparrow .nContent \leftarrow Read\_CompactStatement;
     itAxiom: begin end:
     itReservation: aItem \uparrow. nContent \leftarrow Read\_ReservationSegment;
     itSection:;
     itChoice: begin lList \leftarrow new(PList, Init(0));
          while (nState = eStart) \land ((nElName = SegmentKindName[ikImplQualifiedSegm]) \lor (nElName = SegmentKindName[ikImplQualifiedSegmentKindName])
                         SegmentKindName[ikExplQualifiedSegm])) do lList\uparrow.Insert(Read\_VariableSegment);
```

§1143

```
391
       Mizar Parser
                                                                                    PARSING TEXT ITEMS
  NextElementState; lConds \leftarrow nil;
  if nElName = XMLElemName[elProposition] then
    begin lConds \leftarrow new(PList, Init(0));
     while (nState = eStart) \land (nElName = XMLElemName[elProposition]) do
       lConds \uparrow . Insert(Read\_Proposition);
  NextElementState; \ aItem \uparrow.nContent \leftarrow new(ChoiceStatementPtr, Init(lList, lConds,
       Simple Justification Ptr(Read\_Justification)));
  end;
itReconsider: begin lList \leftarrow new(PList, Init(0));
  while (nState = eStart) \land ((nElName = XMLElemName[elEquality]) \lor (nElName = xMLElemName[elEquality]) \lor (nElName = xMLElemName[elEquality])
          XMLElemName[elVariable])) do
    if nElName = XMLElemName[elVariable] then
       lList \uparrow. Insert(new(TypeChangePtr, Init(VariableIdentifier, Read\_Variable, nil)))
    else begin NextElementState; lVar \leftarrow Read\_Variable;
       lList \uparrow. Insert(new(TypeChangePtr, Init(Equating, lVar, Read\_Term))); NextElementState;
  lType \leftarrow Read\_Type; aItem\uparrow.nContent \leftarrow new(TypeChangingStatementPtr, Init(lList, lType,
       Simple Justification Ptr(Read\_Justification)));
  end:
itPrivFuncDefinition: begin lVar \leftarrow Read\_Variable; lList \leftarrow Read\_TypeList;
  aItem \uparrow .nContent \leftarrow new(PrivateFunctorDefinitionPtr, Init(lVar, lList, Read\_Term));
  end;
itPrivPredDefinition: begin lVar \leftarrow Read\_Variable; lList \leftarrow Read\_TypeList;
  aItem \uparrow .nContent \leftarrow new(PrivatePredicateDefinitionPtr, Init(lVar, lList, Read\_Formula));
  end:
itConstantDefinition: begin lVar \leftarrow Read\_Variable;
  aItem \uparrow .nContent \leftarrow new(ConstantDefinitionPtr, Init(lVar, Read\_Term));
itLociDeclaration, itGeneralization: aItem \uparrow.nContent \leftarrow Read\_VariableSegment;
itPerCases: aItem \uparrow. nContent \leftarrow Read\_Justification;
itCaseBlock:;
itCorrCond: begin CorrectnessConditionPtr(aItem \uparrow. nContent) \uparrow. nJustification \leftarrow Read\_Justification;
  end;
itCorrectness: begin NextElementState;
  while (nState = eStart) \land (nElName = ItemName[itCorrectness]) do
     begin NextElementState; include(CorrectnessConditionsPtr(aItem <math>\uparrow .nContent) \uparrow .nConditions,
          Str2CorrectnessKind(GetAttr(XMLAttrName[atCondition]))); NextElementState;
    end;
  NextElementState; CorrectnessConditionPtr(aItem \uparrow. nContent) \uparrow. nJustification \leftarrow Read\_Justification;
  end;
itProperty:
       aItem \uparrow . nContent \leftarrow new(PropertyPtr, Init(lPos, Str2PropertyKind(aShape), Read\_Justification));
itConclusion, itRegularStatement: aItem \uparrow .nContent \leftarrow Read\_RegularStatement(aShape);
itCaseHead, itSupposeHead, itAssumption: if nState = eStart then
    if nElName = AssumptionKindName[SingleAssumption] then
       begin lPos \leftarrow GetAttrPos; NextElementState;
       aItem \uparrow . nContent \leftarrow new(SingleAssumptionPtr, Init(lPos, Read\_Proposition)); NextElementState;
       end
     else if nElName = AssumptionKindName[CollectiveAssumption] then
         begin lPos \leftarrow GetAttrPos; NextElementState;
```

 $aItem \uparrow . nContent \leftarrow new(CollectiveAssumptionPtr, Init(lPos, new(PList, Init(0))));$

392 Parsing text items Mizar Parser $\S1143$

```
NextElementState;
                 while (nState = eStart) \land (nElName = XMLElemName[elProposition]) do
                      Collective Assumption Ptr(aItem \uparrow. nContent) \uparrow. nConditions \uparrow. Insert(Read\_Proposition);
                 NextElementState; NextElementState;
                 end:
itExistentialAssumption: begin aItem\uparrow.nContent \leftarrow new(ExistentialAssumptionPtr, Init(lPos,
            new(PList, Init(0)), new(PList, Init(0)));
    while (nState = eStart) \land ((nElName = SeqmentKindName[ikImplQualifiedSeqm]) \lor (nElName = SeqmentKindName[ikImplQualifiedSeqmentKindName]) \lor (nElName = SeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQualifiedSeqmentKindName[ikImplQua
                 SegmentKindName[ikExplQualifiedSegm])) do
        Existential Assumption Ptr(a Item \uparrow. n Content) \uparrow. n Q Vars \uparrow. Insert(Read\_Variable Segment);
    NextElementState;
    while (nState = eStart) \land (nElName = XMLElemName[elProposition]) do
        Existential Assumption Ptr(a Item \uparrow. nContent) \uparrow. nConditions \uparrow. Insert(Read\_Proposition);
    NextElementState;
    end:
itExemplification: begin \ lVar \leftarrow nil;
    if (nState = eStart) \land (nElName = XMLElemName[elVariable]) then lVar \leftarrow Read\_Variable;
    lTrm \leftarrow \mathbf{nil};
    if nState \neq eEnd then lTrm \leftarrow Read\_Term;
    aItem \uparrow .nContent \leftarrow new(ExamplePtr, Init(lVar, lTrm));
    end;
itDefPred: begin lRedefinition \leftarrow false;
    if (nState = eStart) \land (nElName = XMLElemName[elRedefine]) then
        begin NextElementState; NextElementState; lRedefinition \leftarrow true;
        end:
    lPattern \leftarrow Read\_PredicatePattern; \ aItem \uparrow .nContent \leftarrow new(PredicateDefinitionPtr, Init(lPos, lPos))
            lRedefinition, PredicatePatternPtr(lPattern), Read_Definiens));
    end;
itDefFunc: begin lRedefinition \leftarrow false;
    if (nState = eStart) \land (nElName = XMLElemName[elRedefine]) then
        begin NextElementState; NextElementState; lRedefinition \leftarrow true;
        end:
    lPattern \leftarrow Read\_FunctorPattern; \ lType \leftarrow nil;
    if (nState = eStart) \land (nElName = XMLElemName[elTypeSpecification]) then
        begin NextElementState; lType \leftarrow Read\_Type; NextElementState;
    if aShape = DefiningWayName[dfMeans] then lDef \leftarrow dfMeans
    else if aShape = DefiningWayName[dfEquals] then lDef \leftarrow dfEquals
        else lDef \leftarrow dfEmpty;
    case lDef of
    dfEquals: aItem \uparrow.nContent \leftarrow new(FunctorDefinitionPtr, Init(lPos, lRedefinition,
                 FunctorPatternPtr(lPattern), lType, lDef, Read\_Definiens));
    dfMeans: aItem \uparrow. nContent \leftarrow new(FunctorDefinitionPtr, Init(lPos, lRedefinition), lRedefinition)
                 FunctorPatternPtr(lPattern), lType, lDef, Read_Definiens));
    dfEmpty: aItem\uparrow.nContent \leftarrow new(FunctorDefinitionPtr, Init(lPos, lRedefinition,
                 FunctorPatternPtr(lPattern), lType, lDef, nil));
    endcases;
    end:
itDefMode: begin lRedefinition \leftarrow false;
    if (nState = eStart) \land (nElName = XMLElemName[elRedefine]) then
        begin NextElementState; NextElementState; lRedefinition \leftarrow true;
        end;
```

§1143 Mizar Parser PARSING TEXT ITEMS 393

```
lPattern \leftarrow Read\_ModePattern;
  if (nState = eStart) \land (nElName = ModeDefinitionSortName[defExpandableMode]) then
    begin NextElementState; aItem\uparrow.nContent \leftarrow new(ExpandableModeDefinitionPtr, Init(CurPos,
         ModePatternPtr(lPattern), Read\_Type)); NextElementState;
    end
  else if (nState = eStart) \land (nElName = ModeDefinitionSortName[defStandardMode]) then
       begin NextElementState; lType \leftarrow nil;
       if (nState = eStart) \land (nElName = XMLElemName[elTypeSpecification]) then
         begin NextElementState; lType \leftarrow Read\_Type; NextElementState;
       aItem \uparrow .nContent \leftarrow new(StandardModeDefinitionPtr, Init(CurPos, lRedefinition,
            ModePatternPtr(lPattern), lType, Read\_Definiens)); NextElementState;
       end:
  end:
itDefAttr: begin lRedefinition \leftarrow false;
  if (nState = eStart) \land (nElName = XMLElemName[elRedefine]) then
    begin NextElementState; NextElementState; lRedefinition \leftarrow true;
    end;
  lPattern \leftarrow Read\_AttributePattern; \ aItem \uparrow. nContent \leftarrow new(AttributeDefinitionPtr, Init(CurPos,
       lRedefinition, AttributePatternPtr(lPattern), Read_Definiens));
  end;
itDefStruct: begin NextElementState; lTyps \leftarrow new(PList, Init(0));
  while nState \neq eEnd do lTyps\uparrow.Insert(Read\_Type);
  NextElementState; lPos \leftarrow GetAttrPos; lNr \leftarrow GetIntAttr(XMLAttrName[atNr]);
  NextElementState; lList \leftarrow nil;
  if (nState = eStart) \land (nElName = XMLElemName[elLoci]) then lList \leftarrow Read\_Loci;
  lFields \leftarrow new(PList, Init(0));
  while (nState = eStart) \land (nElName = XMLElemName[elFieldSegment]) do
    begin lFieldSqmPos \leftarrow GetAttrPos; NextElementState; <math>lSels \leftarrow new(PList, Init(0));
    while (nState = eStart) \land (nElName = XMLElemName[elSelector]) do
       \mathbf{begin}\ lSels \uparrow. Insert(new(FieldSymbolPtr, Init(GetAttrPos, GetIntAttr(XMLAttrName[atNr]))));
       NextElementState; NextElementState;
    lFields \uparrow .Insert(new(FieldSegmentPtr, Init(lFieldSgmPos, lSels, Read\_Type))); NextElementState;
    end;
  NextElementState;
  aItem \uparrow .nContent \leftarrow new(StructureDefinitionPtr, Init(lPos, lTyps, lNr, lList, lFields));
itPredSynonym, itPredAntonym, itFuncNotation, itModeNotation, itAttrSynonym, itAttrAntonym: begin
       lPattern \leftarrow Read\_Pattern; \ aItem \uparrow .nContent \leftarrow new(NotationDeclarationPtr, Init(lPos,
       aItem \uparrow . nItemKind, Read\_Pattern, lPattern));
  end:
itCluster: if nState = eStart then
    if nElName = ClusterRegistrationName[ExistentialRegistration] then
       begin lPos \leftarrow GetAttrPos; NextElementState; lList \leftarrow Read\_AdjectiveList;
       aItem \uparrow . nContent \leftarrow new(EClusterPtr, Init(lPos, lList, Read\_Type)); NextElementState;
       end
    else if nElName = ClusterRegistrationName[ConditionalRegistration] then
         begin lPos \leftarrow GetAttrPos; NextElementState; lList \leftarrow Read\_AdjectiveList;
         lCons \leftarrow Read\_AdjectiveList;
         aItem \uparrow .nContent \leftarrow new(CClusterPtr, Init(lPos, lList, lCons, Read\_Type)); NextElementState;
         end
```

394 PARSING TEXT ITEMS Mizar Parser §1143

```
else if nElName = ClusterRegistrationName[FunctorialRegistration] then
                            begin lPos \leftarrow GetAttrPos; NextElementState; lTrm \leftarrow Read\_Term;
                            lCons \leftarrow Read\_AdjectiveList; lType \leftarrow nil;
                            if nState \neq eEnd then lType \leftarrow Read\_Type;
                            aItem \uparrow . nContent \leftarrow new(FClusterPtr, Init(lPos, lTrm, lCons, lType)); NextElementState;
     itIdentify: \mathbf{begin} \ lPattern \leftarrow Read\_Pattern; \ aItem \uparrow. nContent \leftarrow new(IdentifyRegistrationPtr,
                   Init(lPos, Read\_Pattern, lPattern, new(PList, Init(0)));
         while (nState = eStart) \land (nElName = XMLElemName[elLociEquality]) do
              begin lPos \leftarrow GetAttrPos; NextElementState; <math>lLocus \leftarrow Read\_Locus;
              IdentifyRegistrationPtr(aItem\uparrow.nContent)\uparrow.nEqLociList\uparrow.Insert(new(LociEqualityPtr,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,Init(lPos,In
                        lLocus, Read_Locus))); NextElementState;
              end:
         end:
     itPropertyRegistration: begin lPropertySort \leftarrow Str2PropertyKind(aShape);
         case lPropertySort of
         sySethood: begin
                        aItem \uparrow . nContent \leftarrow new(SethoodRegistrationPtr, Init(lPos, lPropertySort, Read\_Type));
              SethoodRegistrationPtr(aItem\uparrow.nContent)\uparrow.nJustification \leftarrow Read\_Justification;
              end:
         endcases;
         end;
     itReduction: \mathbf{begin} \ lTrm \leftarrow Read\_Term;
         aItem \uparrow .nContent \leftarrow new(ReduceRegistrationPtr, Init(lPos, Read\_Term, lTrm));
         end:
     itPragma:;
     endcases;
     end;
1144.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function In WSMizFileObj.Read_TextProper: wsTextProperPtr;
     var lPos: Position:
     begin NextElementState; lPos.Line \leftarrow GetIntAttr(XMLAttrName[atLine]);
     lPos.Col \leftarrow GetIntAttr(XMLAttrName[atCol]); result \leftarrow new(wsTextProperPtr,
               Init(GetAttr(XMLAttrName[atArticleID]), GetAttr(XMLAttrName[atArticleExt]), lPos));
     if nDisplayInformationOnScreen then DisplayLine(result \uparrow. nBlockPos.Line, 0);
     CurPos \leftarrow result \uparrow .nBlockPos;
     if (nState = eStart) \land (nElName = BlockName[blMain]) then
         begin NextElementState;
         while (nState = eStart) \land (nElName = XMLElemName[elItem]) do
              result \uparrow . nItems \uparrow . Insert(Read\_Item);
         end;
     NextElementState:
     end;
```

 $\{1145$ Mizar Parser PARSING TEXT ITEMS 395

```
1145.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function In WSMizFileObj.Read_Block: wsBlockPtr;
    var lPos: Position;
    begin lPos.Line \leftarrow GetIntAttr(XMLAttrName[atLine]);
    lPos.Col \leftarrow GetIntAttr(XMLAttrName[atCol]);
    result \leftarrow new(WSBlockPtr, Init(Str2BlockKind(GetAttr(XMLAttrName[atKind])), lPos));
    if nDisplayInformationOnScreen then DisplayLine(result \uparrow. nBlockPos.Line, 0);
    lPos.Line \leftarrow GetIntAttr(XMLAttrName[atPosLine]);
    lPos.Col \leftarrow GetIntAttr(XMLAttrName[atPosCol]); result \uparrow .nBlockEndPos \leftarrow lPos;
    CurPos \leftarrow result \uparrow .nBlockPos; NextElementState;
    while (nState = eStart) \land (nElName = XMLElemName[elItem]) do result \uparrow .nItems \uparrow .Insert(Read\_Item);
    CurPos \leftarrow result \uparrow .nBlockEndPos; NextElementState;
    end;
1146.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function In WSMizFileObj.Read_Item: wsItemPtr;
    var lStartTaqNbr: integer; lItemKind: ItemKind; lShape: string; lPos: Position;
    begin lItemKind \leftarrow Str2ItemKind(GetAttr(XMLAttrName[atKind]));
    lPos.Line \leftarrow GetIntAttr(XMLAttrName[atLine]); \ lPos.Col \leftarrow GetIntAttr(XMLAttrName[atCol]);
    CurPos \leftarrow lPos;
    if nDisplayInformationOnScreen then DisplayLine(lPos.Line, 0);
    result \leftarrow new(WSItemPtr, Init(lItemKind, lPos)); lPos.Line \leftarrow GetIntAttr(XMLAttrName[atPosLine]);
    lPos.Col \leftarrow GetIntAttr(XMLAttrName[atPosCol]); result \uparrow .nItemEndPos \leftarrow lPos;
    result \uparrow .nContent \leftarrow nil; Read\_ItemContentsAttr(result, lShape); NextElementState; lStartTagNbr \leftarrow 0;
    if nState \neq eEnd then
         begin Read_ItemContents(result, lShape);
         if (nState = eStart) \land (nElName = XMLElemName[elBlock]) then result \uparrow .nBlock \leftarrow Read\_Block
         else if result \uparrow .nContent = nil then
                  begin repeat if nState = eStart then inc(lStartTagNbr)
                       else dec(lStartTagNbr);
                       NextElementState;
                  until ((nState = eEnd) \land (lStartTagNbr = 0)) \lor ((nState = eStart) \land (nElName = eEnd)) \lor ((nState = eEn
                                XMLElemName[elBlock]);
                  if (nState = eStart) \land (nElName = XMLElemName[elBlock]) then result \uparrow .nBlock \leftarrow Read\_Block;
                  end:
         end;
    CurPos \leftarrow lPos; NextElementState;
    end;
1147.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
function Read_WSMizArticle(aFileName : string): wsTextProperPtr;
    var lInFile: InWSMizFilePtr;
    begin InitWSLookupTables; IInFile \leftarrow new(InWSMizFilePtr, OpenFile(aFileName));
    result \leftarrow lInFile \uparrow. Read\_TextProper; dispose(lInFile, Done); DisposeWSLookupTables;
    end;
```

396

Section 21.13. PRETTYPRINTING WSM FILES (DEFERRED)

1148.

```
\langle \text{ Publicly declared types in wsmarticle.pas } 852 \rangle + \equiv
  WSMizarPrinterPtr = \uparrow WSMizarPrinterObj;
  WSMizarPrinterObj = \mathbf{object} (TXTStreamObj)
    nDisplayInformationOnScreen: boolean;
    nIndent: integer; { indenting }
    constructor OpenFile(const aFileName: string);
    destructor Done; virtual;
    procedure Print\_Char(AChar : char);
    procedure Print_NewLine:
    procedure Print_Number(const aNumber: integer);
    procedure Print_String(const aString: string);
    procedure Print_Indent;
    procedure Print_TextProper(aWSTextProper: WSTextProperPtr); virtual;
    procedure Print_Item(aWSItem : WSItemPtr); virtual;
    procedure Print_SchemeNameInSchemeHead(aSch: SchemePtr); virtual;
    procedure Print_Block(aWSBlock: WSBlockPtr); virtual;
    procedure Print_Adjective(aAttr: AdjectiveExpressionPtr); virtual;
     \textbf{procedure} \ \textit{Print\_AdjectiveList}(\textit{aCluster} : \textit{PList}); \ \textit{virtual}; \\
    procedure Print_Variable(aVar : VariablePtr); virtual;
    procedure Print_ImplicitlyQualifiedVariable(aSegm: ImplicitlyQualifiedSegmentPtr); virtual;
    procedure Print_VariableSegment(aSegm: QualifiedSegmentPtr); virtual;
    procedure Print_Type(aTyp: TypePtr); virtual;
    procedure Print_BinaryFormula(aFrm : BinaryFormulaPtr); virtual;
    procedure Print_PrivatePredicativeFormula(aFrm: PrivatePredicativeFormulaPtr); virtual;
    procedure Print_Formula(aFrm : FormulaPtr); virtual;
    procedure Print_OpenTermList(aTrmList: PList); virtual;
    procedure Print_TermList(aTrmList : PList); virtual;
    procedure Print_SimpleTermTerm(aTrm:SimpleTermPtr): virtual;
    procedure Print_PrivateFunctorTerm(aTrm: PrivateFunctorTermPtr); virtual;
    procedure Print_Term(aTrm : TermPtr); virtual;
    procedure Print_TypeList(aTypeList : PList); virtual;
    procedure Print_Label(aLab : LabelPtr); virtual;
    procedure Print_Reference(aRef : LocalReferencePtr); virtual;
    procedure Print_References(aRefs: PList); virtual;
    procedure Print_StraightforwardJustification(aInf: StraightforwardJustificationPtr); virtual;
    procedure Print_SchemeNameInJustification(aInf: SchemeJustificationPtr); virtual;
    procedure Print_SchemeJustification(aInf: SchemeJustificationPtr); virtual;
    procedure Print_Justification(aInf: JustificationPtr; aBlock: wsBlockPtr); virtual;
    procedure Print_Linkage; virtual;
    procedure Print_RegularStatement(aRStm: RegularStatementPtr; aBlock: wsBlockPtr); virtual;
    procedure Print_CompactStatement(aCStm : CompactStatementPtr; aBlock : wsBlockPtr); virtual;
    procedure Print_Proposition(aProp : PropositionPtr); virtual;
    procedure Print_Conditions(aCond : PList):
    procedure Print_AssumptionConditions(aCond : AssumptionPtr); virtual;
    procedure Print_Pattern(aPattern: PatternPtr); virtual;
    procedure Print_Locus(aLocus : LocusPtr); virtual;
    procedure Print_Loci(aLoci : PList); virtual;
    procedure Print_Definiens(aDef : DefiniensPtr); virtual;
    procedure Print_ReservedType(aResType: TypePtr); virtual;
  end;
```

1149. Constructor.

398

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
constructor WSMizarPrinterObj.OpenFile(const aFileName: string);
  begin inherited InitFile (AFileName); rewrite (nFile); nIndent \leftarrow 0;
  nDisplayInformationOnScreen \leftarrow false;
  end:
destructor WSMizarPrinterObj.Done;
  begin close(nFile); inherited Done;
  end;
1150.
        \langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_Char(aChar: char);
  begin write(nFile, aChar);
  end;
1151. \langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_NewLine;
  begin writeln(nFile);
  end;
1152. \langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_Number(const aNumber: integer);
  begin write(nFile, aNumber); Print_Char(´□´);
  end:
1153. The comment is translated from the Polish comment "?? czy na pewno trzeba robic konwersje", so
I may be mistranslating.
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_String(const aString: string);
  var i: integer;
  begin write(nFile, XMLToStr(aString)); { Do you really need to do conversions? }
  Print\_Char(` \Box `);
  end;
1154. \langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_Indent;
  var i: integer:
  begin for i \leftarrow 1 to nIndent do Print\_Char(` \Box `);
  end:
1155. (Implementation for wsmarticle.pas 854) +\equiv
procedure WSMizarPrinterObj.Print_Adjective(aAttr : AdjectiveExpressionPtr);
  begin case aAttr\uparrow.nAdjectiveSort of
  wsAdjective: with AdjectivePtr(aAttr)\uparrow do
       begin if nArgs\uparrow.Count \neq 0 then Print\_TermList(nArgs);
       Print\_String(AttributeName[nAdjectiveSymbol]);
       end;
  wsNegatedAdjective: begin Print_String(TokenName[sy_Non]);
    Print\_Adjective(NegatedAdjectivePtr(aAttr)\uparrow.nArg);
    end:
  endcases;
  end;
```

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_AdjectiveList(aCluster: PList);
  var i: integer;
  begin with aCluster \uparrow do
     for i \leftarrow 0 to Count - 1 do
       begin Print\_Adjective(Items \uparrow [i]);
       end:
  end;
1157. (Implementation for wsmarticle.pas 854) +\equiv
procedure WSMizarPrinterObj.Print_Variable(aVar : VariablePtr);
  begin with aVar\uparrow do
     begin Print\_String(IdentRepr(nIdent));
     end:
  end;
1158.
        \langle Implementation for wsmarticle.pas 854\rangle + \equiv
procedure WSMizarPrinterObj.Print_ImplicitlyQualifiedVariable(aSegm: ImplicitlyQualifiedSegmentPtr);
  begin Print\_Variable(aSegm \uparrow. nIdentifier);
  end:
       \langle Implementation for wsmarticle.pas 854 \rangle + \equiv
1159.
procedure WSMizarPrinterObj.Print_VariableSegment(aSegm: QualifiedSegmentPtr);
  var i: integer;
  begin case aSegm \uparrow .nSegmentSort of
  ik Impl Qualified Segm:\ Print\_Implicitly Qualified Variable (Implicitly Qualified Segment Ptr (a Segm));
  ikExplQualifiedSegm: with ExplicitlyQualifiedSegmentPtr(aSegm) \uparrow do
       begin Print\_Variable(nIdentifiers.Items \uparrow [0]);
       for i \leftarrow 1 to nIdentifiers \uparrow. Count - 1 do
         begin Print\_String(`,`); Print\_Variable(nIdentifiers \uparrow .Items \uparrow [i]);
       Print\_String(TokenName[sy\_Be]); Print\_Type(nType);
       end;
  endcases;
  end;
1160. (Implementation for wsmarticle.pas 854) +\equiv
procedure WSMizarPrinterObj.Print_OpenTermList(aTrmList: PList);
  var i: integer;
  begin if aTrmList \uparrow. Count > 0 then
     begin Print\_Term(aTrmList\uparrow.Items\uparrow[0]);
     for i \leftarrow 1 to aTrmList \uparrow. Count - 1 do
       begin Print\_String(`,`); Print\_Term(aTrmList\uparrow.Items\uparrow[i]);
       end;
     end;
  end;
```

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_TermList(aTrmList: PList);
  var i: integer;
  begin if aTrmList \uparrow. Count > 0 then
    begin Print\_String(`(`); Print\_Term(aTrmList \uparrow .Items \uparrow [0]);
    for i \leftarrow 1 to aTrmList \uparrow. Count - 1 do
      begin Print\_String(`,`); Print\_Term(aTrmList\uparrow.Items\uparrow[i]);
      end:
    Print_String(`)`);
    end;
  end;
        \langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_Type(aTyp: TypePtr);
  begin with aTyp\uparrow do
    begin case aTyp\uparrow.nTypeSort of
    wsStandardType: with StandardTypePtr(aTyp) \uparrow do
        begin if nArqs \uparrow . Count = 0 then Print\_String(ModeName[nModeSymbol])
         else begin Print_String(`(`); Print_String(ModeName[nModeSymbol]);
           Print\_String(TokenName[sy\_Of]); Print\_OpenTermList(nArgs); Print\_String(`)`);
           end;
         end:
    wsStructureType: with StructTypePtr(aTyp)\uparrow do
         begin if nArgs\uparrow.Count = 0 then Print\_String(StructureName[nStructSymbol])
         else begin Print_String('('); Print_String(StructureName[nStructSymbol]);
           Print\_String(TokenName[sy\_Over]); Print\_OpenTermList(nArgs); Print\_String(`)`);
           end;
         end;
    wsClusteredType: with ClusteredTypePtr(aTyp)\uparrow do
        begin Print_AdjectiveList(nAdjectiveCluster); Print_Type(nType);
    wsErrorType: begin end;
    endcases;
    end;
  end;
1163. (Implementation for wsmarticle.pas 854) +\equiv
procedure WSMizarPrinterObj.Print_BinaryFormula(aFrm: BinaryFormulaPtr);
  begin Print_String(´(´); Print_Formula(aFrm↑.nLeftArg);
  case aFrm\uparrow.nFormulaSort of
  wsConjunctiveFormula: Print_String(TokenName[sy_Ampersand]);
  wsDisjunctiveFormula: Print_String(TokenName[sy_Or]);
  wsConditionalFormula: Print_String(TokenName[sy_Implies]);
  wsBiconditionalFormula: Print_String(TokenName[sy_Iff]);
  wsFlexaryConjunctiveFormula: begin Print_String(TokenName[sy_Ampersand]);
    Print\_String(TokenName[sy\_Ellipsis]); Print\_String(TokenName[sy\_Ampersand]);
    end:
  wsFlexaryDisjunctiveFormula: begin Print_String(TokenName[sy_Or]);
    Print\_String(TokenName[sy\_Ellipsis]); Print\_String(TokenName[sy\_Or]);
  endcases; Print_Formula(aFrm↑.nRightArg); Print_String(´)´);
  end;
```

```
1164. ⟨Implementation for wsmarticle.pas 854⟩ +≡
procedure WSMizarPrinterObj.Print_PrivatePredicativeFormula(aFrm : PrivatePredicativeFormulaPtr);
begin with PrivatePredicativeFormulaPtr(aFrm)↑ do
    begin Print_String(IdentRepr(nPredIdNr)); Print_String(´[´); Print_OpenTermList(nArgs);
    Print_String(´]´);
    end;
end;
```

```
1165.
        \langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_Formula(aFrm: FormulaPtr);
  var i: Integer; lNeq: boolean; lFrm: FormulaPtr;
  begin case aFrm\uparrow.nFormulaSort of
  wsNegatedFormula: begin Print_String(TokenName[sy_Not]);
    Print\_Formula(NegativeFormulaPtr(aFrm)\uparrow.nArg);
    end:
  wsConjunctiveFormula, wsDisjunctiveFormula, wsConditionalFormula,
         wsBiconditional Formula, wsFlexaryConjunctiveFormula, wsFlexaryDisjunctiveFormula:
         Print\_BinaryFormula(BinaryFormulaPtr(aFrm));
  wsPredicativeFormula: with PredicativeFormulaPtr(aFrm) \uparrow do
       begin Print_String(`(`);
       if nLeftArgs\uparrow.Count \neq 0 then
         begin Print_OpenTermList(nLeftArgs);
         end;
       Print\_String(PredicateName[nPredNr]);
       if nRightArgs \uparrow. Count \neq 0 then
         begin Print_OpenTermList(nRightArgs);
         end:
       Print_String(`)`);
       end;
  ws \textit{MultiPredicativeFormula}: \textbf{with} \ \textit{MultiPredicativeFormulaPtr}(\textit{aFrm}) \!\!\uparrow \textbf{do}
       begin Print\_String(``(`); lFrm \leftarrow nScraps.Items^{[0]};
       lNeg \leftarrow lFrm\uparrow.nFormulaSort = wsNegatedFormula;
       if lNeg then lFrm \leftarrow NegativeFormulaPtr(lFrm)\uparrow.nArg;
       with PredicativeFormulaPtr(lFrm)↑ do
         begin if nLeftArgs\uparrow.Count \neq 0 then Print\_OpenTermList(nLeftArgs);
         if lNeg then
           begin Print_String(TokenName[sy_Does]); Print_String(TokenName[sy_Not]);
            end:
         Print\_String(PredicateName[nPredNr]);
         if nRightArgs \uparrow . Count \neq 0 then Print\_OpenTermList(nRightArgs);
         end:
       for i \leftarrow 1 to nScraps.Count - 1 do
         begin lFrm \leftarrow nScraps.Items \uparrow [i]; lNeq \leftarrow lFrm \uparrow .nFormulaSort = wsNeqatedFormula;
         if lNeq then lFrm \leftarrow NegativeFormulaPtr(lFrm)\uparrow.nArq;
         with RightSideOfPredicativeFormulaPtr(lFrm) \uparrow do
           begin if lNeq then
              begin Print_String(TokenName[sy_Does]); Print_String(TokenName[sy_Not]);
              end;
            Print\_String(PredicateName[nPredNr]);
           if nRightArgs \uparrow . Count \neq 0 then Print\_OpenTermList(nRightArgs);
            end:
         end;
       Print_String(`)`);
       end;
  wsPrivatePredicateFormula: Print\_PrivatePredicativeFormula(PrivatePredicativeFormulaPtr(aFrm));
  wsAttributiveFormula: with AttributiveFormulaPtr(aFrm) \uparrow do
       begin Print_String(`(`); Print_Term(nSubject); Print_String(TokenName[sy_Is]);
       Print_AdjectiveList(nAdjectives); Print_String(`)`);
       end;
  wsQualifyingFormula: with QualifyingFormulaPtr(aFrm)\uparrow do
```

```
begin Print_String(`(`); Print_Term(nSubject); Print_String(TokenName[sy_Is]);
      Print_Type(nType); Print_String(`)`);
  wsUniversalFormula: with QuantifiedFormulaPtr(aFrm) \uparrow  do
      begin Print_String(`(`); Print_String(TokenName[sy_For]);
      Print\_VariableSegment(QuantifiedFormulaPtr(aFrm)\uparrow.nSegment);
      Print\_String(TokenName[sy\_Holds]); Print\_Formula(QuantifiedFormulaPtr(aFrm)\uparrow.nScope);
      Print_String(`)`);
      end;
  wsExistentialFormula: with QuantifiedFormulaPtr(aFrm) \uparrow do
      begin Print_String(`(`); Print_String(TokenName[sy_Ex]);
      Print\_VariableSegment(QuantifiedFormulaPtr(aFrm)\uparrow.nSegment);\ Print\_String(TokenName[sy\_St]);
      Print\_Formula(QuantifiedFormulaPtr(aFrm)\uparrow.nScope); Print\_String(`)`);
  wsContradiction: begin Print_String(TokenName[sy_Contradiction]);
  wsThesis: begin Print_String(TokenName[sy_Thesis]);
  wsErrorFormula: begin end;
  endcases;
  end;
1166. (Implementation for wsmarticle.pas 854) +\equiv
procedure WSMizarPrinterObj.Print_SimpleTermTerm(aTrm:SimpleTermPtr);
  begin Print\_String(IdentRepr(SimpleTermPtr(aTrm)\uparrow.nIdent));
  end;
1167. (Implementation for wsmarticle.pas 854) +\equiv
procedure WSMizarPrinterObj.Print_PrivateFunctorTerm(aTrm:PrivateFunctorTermPtr);
  begin Print_String(IdentRepr(aTrm\f\.nFunctorIdent)); Print_String(\(`(')\);
  Print\_OpenTermList(aTrm \uparrow .nArgs); Print\_String(`)`);
  end;
```

404

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_Term(aTrm: TermPtr);
  var i, j: integer; lPrintWhere: boolean;
  begin case aTrm \uparrow .nTermSort of
  wsPlaceholderTerm: begin Print\_Char(`\$`); Print\_Number(PlaceholderTermPtr(aTrm)\uparrow.nLocusNr);
  wsSimpleTerm: begin Print_SimpleTermTerm(SimpleTermPtr(aTrm));
    end:
  wsNumeralTerm: begin Print_Number(NumeralTermPtr(aTrm)\uparrow.nValue);
  wsInfixTerm: with InfixTermPtr(aTrm)\uparrow \mathbf{do}
      begin Print_String(`(`);
      if nLeftArgs\uparrow.Count \neq 0 then
        begin Print_TermList(nLeftArgs);
         end:
       Print\_String(FunctorName[nFunctorSymbol]);
      if nRightArgs \uparrow. Count \neq 0 then
         begin Print_TermList(nRightArgs);
        end:
       Print_String(`)`);
      end;
  wsCircumfixTerm: with CircumfixTermPtr(aTrm)\uparrow do
       begin Print\_String(LeftBracketName[nLeftBracketSymbol]); Print\_OpenTermList(nArgs);
       Print\_String(RightBracketName[nRightBracketSymbol]);
      end:
  wsPrivateFunctorTerm: Print\_PrivateFunctorTerm(PrivateFunctorTermPtr(aTrm));
  wsAggregateTerm: with AggregateTermPtr(aTrm)\uparrow do
      begin Print_String(StructureName[nStructSymbol]);
       Print\_String(TokenName[sy\_StructLeftBracket]); Print\_OpenTermList(nArgs);
       Print_String(TokenName[sy_StructRightBracket]);
  wsSelectorTerm: with SelectorTermPtr(aTrm)\uparrow do
      begin Print_String(`(`); Print_String(TokenName[sy_The]);
       Print\_String(SelectorName[nSelectorSymbol]); Print\_String(TokenName[sy\_Of]);
       Print_Term(nArg); Print_String(`)`);
  wsInternalSelectorTerm: with InternalSelectorTermPtr(aTrm) \uparrow  do
       \mathbf{begin} \ Print\_String(TokenName[sy\_The]); \ Print\_String(SelectorName[nSelectorSymbol]);
      end:
  wsForgetfulFunctorTerm: with ForgetfulFunctorTermPtr(aTrm) \uparrow do
       begin Print_String(`(`); Print_String(TokenName[sy_The]);
       Print\_String(StructureName[nStructSymbol]); Print\_String(TokenName[sy\_Of]);
       Print\_Term(nArg); Print\_String(`)`);
      end:
  wsInternalForgetfulFunctorTerm: with InternalForgetfulFunctorTermPtr(aTrm) \uparrow do
      begin Print_String(`(`); Print_String(TokenName[sy_The]);
       Print_String(StructureName[nStructSymbol]); Print_String(`)`);
  wsFraenkelTerm: with FraenkelTermPtr(aTrm) \uparrow do
       begin Print_String(`{`); Print_Term(nSample);
      if nPostqualification \uparrow. Count > 0 then
        begin lPrintWhere \leftarrow true;
```

```
for i \leftarrow 0 to nPostgualification \uparrow. Count - 1 do
          case QualifiedSegmentPtr(nPostqualification \uparrow .Items \uparrow [i]) \uparrow .nSegmentSort of
          ikImplQualifiedSeqm: with ImplicitlyQualifiedSeqmentPtr(nPostqualification\uparrow.Items\uparrow[i])\uparrow do
               begin Print_String(TokenName[sy_Where]); Print_Variable(nIdentifier);
               end:
          ikExplQualifiedSegm: with ExplicitlyQualifiedSegmentPtr(nPostqualification \uparrow. Items \uparrow [i]) \uparrow do
               begin if lPrintWhere then
                 begin Print\_String(TokenName[sy\_Where]); lPrintWhere \leftarrow false;
                 end:
               Print\_Variable(nIdentifiers.Items \uparrow [0]);
               for j \leftarrow 1 to nIdentifiers \uparrow. Count - 1 do
                 begin Print\_String(`,`); Print\_Variable(nIdentifiers \uparrow. Items \uparrow [j]);
                 end:
               Print\_String(TokenName[sy\_Is]); Print\_Type(nType);
               if i < nPostqualification \uparrow. Count - 1 then Print\_String( `, `);
               end;
          endcases;
       end;
     Print_String(':'); Print_Formula(nFormula); Print_String('});
wsSimpleFraenkelTerm: with SimpleFraenkelTermPtr(aTrm) \uparrow do
     begin Print_String(`(`); Print_String(TokenName[sy_The]); Print_String(TokenName[sy_Set]);
     Print\_String(TokenName[sy\_Of]); Print\_String(TokenName[sy\_All]); Print\_Term(nSample);
     if nPostqualification \uparrow. Count > 0 then
       begin lPrintWhere \leftarrow true;
       for i \leftarrow 0 to nPostqualification \uparrow. Count - 1 do
          case QualifiedSegmentPtr(nPostqualification \uparrow .Items \uparrow [i]) \uparrow .nSegmentSort of
          ikImplQualifiedSegm: with ImplicitlyQualifiedSegmentPtr(nPostqualification \uparrow. Items \uparrow [i]) \uparrow do
               begin Print_String(TokenName[sy_Where]); Print_Variable(nIdentifier);
               end:
          ikExplQualifiedSeqm: with ExplicitlyQualifiedSeqmentPtr(nPostqualification \uparrow . Items \uparrow [i]) \uparrow do
               begin if lPrintWhere then
                 begin Print\_String(TokenName[sy\_Where]); lPrintWhere \leftarrow false;
                 end;
               Print\_Variable(nIdentifiers.Items \uparrow [0]);
               for j \leftarrow 1 to nIdentifiers \uparrow. Count - 1 do
                 begin Print\_String(`,`); Print\_Variable(nIdentifiers \uparrow. Items \uparrow [j]);
               Print\_String(TokenName[sy\_Is]); Print\_Type(nType);
               if i < nPostqualification \uparrow. Count - 1 then Print\_String( `, `);
               end:
          endcases;
       end:
     Print\_String(`)`);
     end;
wsQualificationTerm: with QualifiedTermPtr(aTrm) \uparrow do
     \textbf{begin} \ \textit{Print\_String(``(`)'}; \ \textit{Print\_Term(nSubject)}; \ \textit{Print\_String(TokenName[sy\_Qua])}; \\
     Print\_Type(nQualification); Print\_String(`)`);
    end:
wsExactlyTerm: with ExactlyTermPtr(aTrm)\uparrow do
     begin Print_Term(nSubject); Print_String(TokenName[sy_Exactly]);
     end;
```

```
wsGlobalChoiceTerm: begin Print_String(`(`); Print_String(TokenName[sy_The]);
    Print\_Type(ChoiceTermPtr(aTrm)\uparrow.nChoiceType); Print\_String(`)`);
  wsItTerm: begin Print_String(TokenName[sy_It]);
    end;
  ws Error Term:\\
  endcases:
  end:
        \langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_TypeList(aTypeList : PList);
  var i: integer;
  begin if aTypeList \uparrow. Count > 0 then
    begin Print_Type(aTypeList^{\uparrow}.Items^{\uparrow}[0]);
    for i \leftarrow 1 to aTypeList \uparrow. Count - 1 do
       begin Print\_String(`,`); Print\_Type(aTypeList\uparrow.Items\uparrow[i]);
       end;
    end;
  end;
1170. (Implementation for wsmarticle.pas 854) +\equiv
procedure WSMizarPrinterObj.Print_Label(aLab : LabelPtr);
  begin if (aLab \neq nil) \land (aLab.nLabelIdNr > 0) then
    begin Print_String(IdentRepr(aLab\frac{1}{2}.nLabelIdNr)); Print_String(':');
    end;
  end;
1171. (Implementation for wsmarticle.pas 854) +\equiv
procedure WSMizarPrinterObj.Print_Proposition(aProp : PropositionPtr);
  begin Print\_Label(aProp\uparrow.nLab); Print\_Formula(aProp\uparrow.nSentence);
  end;
       \langle Implementation for wsmarticle.pas 854 \rangle + \equiv
{\bf procedure}\ WSMizarPrinterObj.Print\_CompactStatement(aCStm:CompactStatementPtr;
         aBlock: wsBlockPtr);
  begin with aCStm\uparrow do
    begin Print_Proposition(nProp); Print_Justification(nJustification, aBlock);
    end;
  end;
1173. (Implementation for wsmarticle.pas 854) +\equiv
procedure WSMizarPrinterObj.Print_Linkage;
  begin Print_String(TokenName[sy_Then]);
  end;
```

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_RegularStatement(aRStm: RegularStatementPtr;
         aBlock: wsBlockPtr);
  var i: integer;
  begin case aRStm\uparrow.nStatementSort of
  stDiffuseStatement: begin Print\_Label(DiffuseStatementPtr(aRStm)\uparrow.nLab); Print\_Block(aBlock);
    end:
  stCompactStatement: begin
           if (CompactStatementPtr(aRStm)\uparrow.nJustification\uparrow.nInfSort = infStraightforwardJustification) \land
            StraightforwardJustificationPtr(CompactStatementPtr(aRStm) \uparrow .nJustification) \uparrow .nLinked then
       begin Print_Linkage;
       end:
    Print_CompactStatement(CompactStatementPtr(aRStm), aBlock);
    end:
  stIterativeEquality: begin
            if (CompactStatementPtr(aRStm)\uparrow.nJustification\uparrow.nInfSort = infStraightforwardJustification) \land
            StraightforwardJustificationPtr(CompactStatementPtr(aRStm) \uparrow .nJustification) \uparrow .nLinked then
       begin Print_Linkage;
       end:
    Print_CompactStatement(CompactStatementPtr(aRStm), nil);
    with IterativeEqualityPtr(aRStm)\uparrow do
       for i \leftarrow 0 to nIterSteps \uparrow. Count - 1 do
         with IterativeStepPtr(nIterSteps\uparrow.Items\uparrow[i])\uparrow do
            begin Print_NewLine; Print_String(TokenName[sy_DotEquals]); Print_Term(nTerm);
            Print_Justification(nJustification, nil);
            end:
    end;
  endcases;
  end;
        \langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_Reference(aRef : LocalReferencePtr);
  begin Print\_String(IdentRepr(aRef \uparrow .nLabId));
  end;
```

```
1176.
        \langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_References(aRefs: PList);
  var i: integer;
  begin for i \leftarrow 0 to aRefs \uparrow . Count - 1 do
    with ReferencePtr(aRefs\uparrow.Items\uparrow[i])\uparrow do
       begin case nRefSort of
       LocalReference: begin Print_Reference(aRefs\uparrow.Items\uparrow[i]);
         end:
       TheoremReference: begin
              Print\_String(MMLIdentifierName[TheoremReferencePtr(aRefs\uparrow.Items\uparrow[i])\uparrow.nArticleNr]);
         Print\_String(`:`); Print\_Number(TheoremReferencePtr(aRefs\uparrow.Items\uparrow[i])\uparrow.nTheoNr);
         end;
       DefinitionReference: begin
               Print\_String(MMLIdentifierName[DefinitionReferencePtr(aRefs\uparrow.Items\uparrow[i])\uparrow.nArticleNr]);
         Print_String(`:`); Print_String(`def');
         Print\_Number(DefinitionReferencePtr(aRefs\uparrow.Items\uparrow[i])\uparrow.nDEfNr);
         end;
       endcases;
       if i < aRefs \uparrow. Count - 1 then Print\_String(`,`);
       end:
  end;
1177. \langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_StraightforwardJustification(aInf:StraightforwardJustificationPtr);
  begin with aInf \uparrow do
    begin if nReferences \uparrow. Count \neq 0 then
       begin Print_String(TokenName[sy_By]); Print_References(nReferences);
       end;
    end;
  end;
        \langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_SchemeNameInJustification(aInf: SchemeJustificationPtr);
  begin Print\_String(IdentRepr(aInf \uparrow .nSchemeIdNr));
  end;
1179. \langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_SchemeJustification(aInf:SchemeJustificationPtr);
  begin with aInf↑ do
    begin Print_String(TokenName[sy_From]);
    if nSchFileNr > 0 then
       begin Print_String(MMLIdentifierName[nSchFileNr]); Print_String(`:`); Print_String(`sch`);
       Print_Number(nSchemeIdNr);
       end
    else if nSchemeIdNr > 0 then Print\_SchemeNameInJustification(aInf);
    if nReferences \uparrow. Count > 0 then
       begin Print_String(`(`); Print_References(nReferences); Print_String(`)`);
       end:
    end;
  end;
```

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_Justification(aInf: JustificationPtr; aBlock: wsBlockPtr);
  begin case aInf \uparrow .nInfSort of
  infStraightforwardJustification: Print\_StraightforwardJustification(StraightforwardJustificationPtr(aInf));
  infSchemeJustification: Print\_SchemeJustification(SchemeJustificationPtr(aInf));
  infError, infSkippedProof: begin end;
  infProof: Print_Block(aBlock);
  endcases;
  end:
1181. (Implementation for wsmarticle.pas 854) +\equiv
procedure WSMizarPrinterObj.Print_Conditions(aCond : PList);
  var i: integer;
  begin Print\_String(TokenName[sy\_That]); Print\_NewLine; Print\_Proposition(aCond \cdot. Items \cdot[0]);
  for i \leftarrow 1 to aCond \uparrow . Count - 1 do
    begin Print\_String(TokenName[sy\_And]); Print\_NewLine; Print\_Proposition(aCond \uparrow .Items \uparrow [i]);
    end;
  end;
        \langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure \ WSMizarPrinterObj.Print\_AssumptionConditions(aCond: AssumptionPtr);
  begin case aCond\uparrow.nAssumptionSort of
  Single Assumption: \mathbf{begin} \ Print\_Proposition(Single AssumptionPtr(aCond)\uparrow.nProp);
  Collective Assumption: begin Print_Conditions (Collective Assumption Ptr(aCond)\uparrow.nConditions);
    end:
  endcases;
  end;
1183. (Implementation for wsmarticle.pas 854) +\equiv
procedure WSMizarPrinterObj.Print_Locus(aLocus: LocusPtr);
  begin with aLocus↑ do
    begin Print\_String(IdentRepr(nVarId));
    end;
  end;
1184. (Implementation for wsmarticle.pas 854) +\equiv
procedure WSMizarPrinterObj.Print_Loci(aLoci : PList);
  var i: integer;
  begin if (aLoci = nil) \lor (aLoci \uparrow. Count = 0) then
  else begin Print\_Locus(aLoci\uparrow.Items\uparrow[0]);
    for i \leftarrow 1 to aLoci \uparrow. Count - 1 do
       begin Print\_String(`,`); Print\_Locus(aLoci\uparrow.Items\uparrow[i]);
       end;
    end;
  end;
```

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_Pattern(aPattern: PatternPtr);
  begin case aPattern \uparrow . nPatternSort of
  itDefPred: with PredicatePatternPtr(aPattern)↑ do
       begin Print_Loci(nLeftArgs); Print_String(PredicateName[nPredSymbol]); Print_Loci(nRightArgs);
  itDefFunc: begin case FunctorPatternPtr(aPattern) \upsi.nFunctKind of
    InfixFunctor: with InfixFunctorPatternPtr(aPattern)↑ do
         begin if (nLeftArgs \neq nil) \land (nLeftArgs \uparrow. Count > 1) then Print\_String(``(`);
         Print\_Loci(nLeftArgs);
         if (nLeftArgs \neq nil) \land (nLeftArgs \uparrow. Count > 1) then Print\_String(`)`);
         Print\_String(FunctorName[nOperSymb]);
         if (nRightArgs \neq nil) \land (nRightArgs \uparrow. Count > 1) then Print\_String(`(`);
         Print\_Loci(nRightArgs);
         if (nRightArgs \neq nil) \land (nRightArgs \uparrow. Count > 1) then Print\_String( `) `);
         end;
    CircumfixFunctor: with CircumfixFunctorPatternPtr(aPattern)\uparrow do
         begin Print\_String(LeftBracketName[nLeftBracketSymb]); Print\_Loci(nArgs);
         Print\_String(RightBracketName[nRightBracketSymb]);
         end:
    endcases;
    end;
  itDefMode: with ModePatternPtr(aPattern)↑ do
       begin Print_String(ModeName[nModeSymbol]);
      if (nArgs \neq nil) \land (nArgs \uparrow. Count > 0) then
         begin Print_String(TokenName[sy_Of]); Print_Loci(nArgs);
         end;
       end;
  itDefAttr: with AttributePatternPtr(aPattern)↑ do
       begin Print_Locus(nArg); Print_String(TokenName[sy_Is]); Print_Loci(nArgs);
       Print\_String(AttributeName[nAttrSymbol]);
       end;
  endcases;
  end;
```

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_Definiens(aDef : DefiniensPtr);
  var i: integer;
  begin if aDef \neq nil then
    with DefiniensPtr(aDef)\uparrow \mathbf{do}
       begin case nDefSort of
       Simple Definiens: begin if (nDefLabel \neq nil) \land (nDefLabel \uparrow .nLabel IdNr > 0) then
           begin Print_String(`:`); Print_Label(nDefLabel);
           end:
         with SimpleDefiniensPtr(aDef)\uparrow, nExpression\uparrow do
           case nExprKind of
            exTerm: Print_Term(TermPtr(nExpr));
            exFormula: Print_Formula(FormulaPtr(nExpr));
           endcases:
         end;
       Conditional Definiens: begin if (nDefLabel \neq nil) \land (nDefLabel \uparrow .nLabelIdNr > 0) then
           begin Print_String(`:`); Print_Label(nDefLabel);
           end;
         with ConditionalDefiniensPtr(aDef) \uparrow do
           begin for i \leftarrow 0 to nConditionalDefiniensList \uparrow. Count - 1 do
              begin with PartDefPtr(nConditionalDefiniensList\uparrow.Items\uparrow[I])\uparrow do
                begin with nPartDefiniens↑ do
                  case nExprKind of
                   exTerm: Print_Term(TermPtr(nExpr));
                   exFormula: Print_Formula(FormulaPtr(nExpr));
                  endcases:
                Print\_String(TokenName[sy\_If]); Print\_Formula(nGuard);
                end;
              if (i \ge 0) \land (i < nConditionalDefiniensList \uparrow. Count - 1) then
                begin Print_String(´, ´); Print_NewLine;
                end:
              end;
           if nOtherwise \neq nil then
              with nOtherwise \uparrow do
                begin Print_String(TokenName[sy_Otherwise]);
                case nExprKind of
                exTerm: Print\_Term(TermPtr(nExpr));
                exFormula: Print_Formula(FormulaPtr(nExpr));
                endcases;
                end;
           end;
         end;
       end;
       endcases;
  end;
```

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_Block(aWSBlock: WSBlockPtr);
  var i, lIndent: integer;
  begin with aWSBlock \uparrow do
    \mathbf{begin} \ \mathit{lIndent} \leftarrow \mathit{nIndent}; \ \mathit{Print\_NewLine}; \ \mathit{Print\_Indent};
    case nBlockKind of
    blDiffuse: begin Print_String(TokenName[sy_Now]); Print_NewLine;
    blHereby: begin Print_String(TokenName[sy_Now]); Print_NewLine;
    blProof: begin Print_String(TokenName[sy_Proof]); Print_NewLine;
    blDefinition: begin Print_String(TokenName[sy_Definition]); Print_NewLine;
    blNotation: begin Print_String(TokenName[sy_Notation]); Print_NewLine;
    blRegistration: begin Print_String(TokenName[sy_Registration]); Print_NewLine;
       end;
    blCase: Print\_String(TokenName[sy\_Case]);
    blSuppose: Print_String(TokenName[sy_Suppose]);
    blPublicScheme\colon;
    endcases;
    for i \leftarrow 0 to nItems \uparrow. Count - 1 do
       begin Print\_Item(nItems\uparrow.Items\uparrow[i]);
    nIndent \leftarrow lIndent; Print\_Indent; Print\_String(TokenName[sy\_End]);
    end;
  end;
1188. (Implementation for wsmarticle.pas 854) +\equiv
procedure WSMizarPrinterObj.Print_TextProper(aWSTextProper: WSTextProperPtr);
  var i: integer;
  begin with aWSTextProper \uparrow do
    begin for i \leftarrow 0 to nItems \uparrow. Count - 1 do Print\_Item(nItems \uparrow. Items \uparrow [i]);
    end;
  end;
1189. (Implementation for wsmarticle.pas 854) +\equiv
procedure WSMizarPrinterObj.Print_ReservedType(aResType: TypePtr);
  begin Print_{-}Type(aResType);
  end;
        \langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_SchemeNameInSchemeHead(aSch:SchemePtr);
  begin Print\_String(IdentRepr(aSch \uparrow .nSchemeIdNr));
  end;
```

```
\langle Implementation for wsmarticle.pas 854 \rangle + \equiv
procedure WSMizarPrinterObj.Print_Item(aWSItem: WSItemPtr);
  var i, j, lIndent: integer;
  begin with aWSItem \uparrow do
     begin CurPos \leftarrow nItemPos;
     if nDisplayInformationOnScreen then DisplayLine(CurPos.Line, ErrorNbr);
     case nItemKind of
     itDefinition: begin Print_Block(nBlock); Print_String(';'); Print_NewLine;
     itSchemeBlock: begin Print_Block(nBlock); Print_String(';'); Print_NewLine;
       end;
     itSchemeHead: with SchemePtr(nContent) \uparrow do
         begin Print_String(TokenName[sy_Scheme]);
          Print_SchemeNameInSchemeHead(SchemePtr(nContent)); Print_String(`{`);
         for j \leftarrow 0 to nSchemeParams \uparrow. Count - 1 do
            begin case SchemeSegmentPtr(nSchemeParams\uparrow.Items\uparrow[j])\uparrow.nSegmSort of
            PredicateSegment: with PredicateSegmentPtr(nSchemeParams \uparrow .Items \uparrow [j]) \uparrow do
                 begin Print\_Variable(nVars\uparrow.Items\uparrow[0]);
                 for i \leftarrow 1 to nVars \uparrow. Count - 1 do
                   begin Print\_String(`,`); Print\_Variable(nVars\uparrow.Items\uparrow[i]);
                   end:
                 Print_String(`[`); Print_TypeList(nTypeExpList); Print_String(`]`);
                 end;
            FunctorSegment: with FunctorSegmentPtr(nSchemeParams\uparrow.Items\uparrow[j])\uparrow do
                 begin Print\_Variable(nVars\uparrow.Items\uparrow[0]);
                 for i \leftarrow 1 to nVars.Count - 1 do
                   begin Print\_String(`,`); Print\_Variable(nVars\uparrow.Items\uparrow[i]);
                   end:
                 Print_String(`(`); Print_TypeList(nTypeExpList); Print_String(`)`);
                 Print_String(TokenName[sy_Arrow]); Print_Type(nSpecification);
                 end:
            endcases:
            if (j > 0) \land (j < nSchemeParams \uparrow. Count - 1) then Print\_String(\uparrow, \uparrow);
          Print_String(`\formula(nSchemeConclusion); Print_Newline; Print_Formula(nSchemeConclusion);
          Print_NewLine;
         if (nSchemePremises \neq nil) \land (nSchemePremises \uparrow. Count > 0) then
            begin Print_String(TokenName[sy_Provided]);
            Print\_Proposition(nSchemePremises \uparrow.Items \uparrow [0]);
            for i \leftarrow 1 to nSchemePremises \uparrow. Count - 1 do
              begin Print_String(TokenName[sy_And]); Print_NewLine;
               Print\_Proposition(nSchemePremises \uparrow .Items \uparrow [i]);
              end:
            end;
          Print_String(TokenName[sy_Proof]); Print_NewLine;
     itTheorem: with CompactStatementPtr(nContent) \uparrow do
         begin Print\_NewLine: nIndent \leftarrow 0: Print\_String(TokenName[sy\_Theorem]):
          Print\_Label(nProp\uparrow.nLab); Print\_NewLine; nIndent \leftarrow 2; Print\_Indent;
          Print\_Formula(nProp\uparrow.nSentence); nIndent \leftarrow 0; Print\_Justification(nJustification, nBlock);
          Print_String(';'); Print_NewLine;
         end;
```

414

```
itAxiom: begin end;
itReservation: with ReservationSegmentPtr(nContent) \uparrow do
    begin Print_NewLine; Print_String(TokenName[sy_reserve]);
    Print\_Variable(nIdentifiers.Items\uparrow[0]);
    for i \leftarrow 1 to nIdentifiers \uparrow. Count - 1 do
       begin Print\_String(`,`); Print\_Variable(nIdentifiers\uparrow.Items\uparrow[i]);
    Print_String(TokenName[sy_For]); Print_ReservedType(nResType); Print_String(';');
    Print_NewLine;
itSection: begin Print_NewLine; Print_String(TokenName[sy_Begin]); Print_NewLine;
itRegularStatement: begin Print_RegularStatement(RegularStatementPtr(nContent), nBlock);
  Print_String(`;`); Print_NewLine;
  end:
itChoice: with ChoiceStatementPtr(nContent)↑ do
    begin if (nJustification \uparrow .nInfSort = infStraightforwardJustification) <math>\land
            StraightforwardJustificationPtr(nJustification) \uparrow .nLinked then
       begin Print_Linkage;
       end:
    Print\_String(TokenName[sy\_Consider]); Print\_VariableSegment(nQualVars\uparrow.Items\uparrow[0]);
    for i \leftarrow 1 to nQualVars \uparrow. Count - 1 do
       begin Print\_String(`,`); Print\_VariableSegment(nQualVars \uparrow .Items \uparrow [i]);
    if (nConditions \neq nil) \land (nConditions \uparrow. Count > 0) then
       begin Print_String(TokenName[sy_Such]); Print_Conditions(nConditions);
    Print_Justification(nJustification, nil); Print_String(';'); Print_NewLine;
itReconsider: with TypeChangingStatementPtr(nContent) \uparrow do
    begin if (nJustification \uparrow .nInfSort = infStraightforwardJustification) <math>\land
            StraightforwardJustificationPtr(nJustification) \uparrow .nLinked then
       begin Print_Linkage;
       end;
    Print_String(TokenName[sy_Reconsider]);
    for i \leftarrow 0 to nTypeChangeList \uparrow. Count - 1 do
       begin case TypeChanqePtr(nTypeChanqeList\uparrow.Items\uparrow[i])\uparrow.nTypeChanqeKind of
       Equating: begin Print_Variable(TypeChangePtr(nTypeChangeList^1.Items^{[i]})^1.nVar);
         Print\_String(`=`); Print\_Term(TypeChangePtr(nTypeChangeList \uparrow .Items \uparrow [i]) \uparrow .nTermExpr);
       Variable Identifier: \mathbf{begin} \ Print\_Variable (Type Change Ptr(nType Change List\_Items \uparrow [i]) \uparrow . nVar);
         end:
       endcases:
       if (i \ge 0) \land (i < nTypeChangeList \uparrow. Count - 1) then Print\_String( `, `);
    Print\_String(TokenName[sy\_As]); Print\_Type(nTypeExpr);
    Print_Justification(nJustification, nil); Print_String(';'); Print_NewLine;
itPrivFuncDefinition: with PrivateFunctorDefinitionPtr(nContent) \uparrow do
    begin Print_String(TokenName[sy_DefFunc]); Print_Variable(nFuncId); Print_String(`(`);
    Print_TypeList(nTypeExpList); Print_String(`)`); Print_String(`=`); Print_Term(nTermExpr);
    Print_String(';'); Print_NewLine;
```

```
end;
itPrivPredDefinition: with PrivatePredicateDefinitionPtr(nContent) \uparrow do
    begin Print_String(TokenName[sy_DefPred]); Print_Variable(nPredId); Print_String(`[`);
    Print_TypeList(nTypeExpList); Print_String(`]`); Print_String(TokenName[sy_Means]);
    Print_Formula(nSentence); Print_String(';'); Print_NewLine;
    end;
itConstantDefinition: with ConstantDefinitionPtr(nContent) \uparrow do
    begin Print_String(TokenName[sy_Set]); Print_Variable(nVarId); Print_String('=');
    Print_Term(nTermExpr); Print_String(';'); Print_NewLine;
itLociDeclaration, itGeneralization: begin Print_String(TokenName[sy_Let]);
  Print_VariableSegment(QualifiedSegmentPtr(nContent)); Print_String(';'); Print_NewLine;
itAssumption: begin Print_String(TokenName[sy_Assume]);
  Print_AssumptionConditions(AssumptionPtr(nContent)); Print_String(´;´); Print_NewLine;
  end;
itExistentialAssumption: with ExistentialAssumptionPtr(nContent) \uparrow do
    begin Print\_String(TokenName[sy\_Given]); Print\_VariableSegment(nQVars <math>\uparrow. Items \uparrow [0]);
    for i \leftarrow 1 to nQVars \uparrow. Count - 1 do
      begin Print\_String(`, `); Print\_VariableSegment(nQVars \uparrow .Items \uparrow [i]);
      end;
    Print_String(TokenName[sy_Such]); Print_String(TokenName[sy_That]); Print_NewLine;
    Print\_Proposition(nConditions \uparrow .Items \uparrow [0]);
    for i \leftarrow 1 to nConditions \uparrow. Count - 1 do
      begin Print_String(TokenName[sy_And]); Print_NewLine;
      Print\_Proposition(nConditions \uparrow .Items \uparrow [i]);
    Print_String(';'); Print_NewLine;
itExemplification: with ExamplePtr(nContent)↑ do
    begin Print_String(TokenName[sy_Take]);
    if nVarId \neq nil then
      begin Print_Variable(nVarId);
      if nTermExpr \neq nil then
         begin Print_String('=');
         end;
      end;
    if nTermExpr \neq nil then Print\_Term(nTermExpr);
    Print_String(`;`); Print_NewLine;
    end;
itPerCases: begin if (JustificationPtr(nContent)\uparrow.nInfSort =
         infStraightforwardJustification) \land StraightforwardJustificationPtr(nContent) \uparrow .nLinked then
    begin Print_Linkage;
    end:
  Print\_String(TokenName[sy\_Per]); Print\_String(TokenName[sy\_Cases]);
  Print_Justification(JustificationPtr(nContent), nil); Print_String(´;´); Print_NewLine;
  end:
itConclusion: begin Print_String(TokenName[sy_Thus]);
  Print_RegularStatement(RegularStatementPtr(nContent), nBlock); Print_String(';');
  Print_NewLine;
  end;
itCaseBlock: begin Print_Block(nBlock); Print_String(';'); Print_NewLine;
```

416

end;

```
itCaseHead, itSupposeHead: begin Print\_AssumptionConditions(AssumptionPtr(nContent));
  Print_String(`;`); Print_NewLine;
  end;
itCorrCond: begin
      Print\_String(CorrectnessName[CorrectnessConditionPtr(nContent) \uparrow .nCorrCondSort]);
  Print\_Justification(CorrectnessConditionPtr(nContent) \uparrow nJustification, nBlock); Print\_String(';');
  Print_NewLine:
  end;
itCorrectness: begin Print_String(TokenName[sy_Correctness]);
  Print\_Justification(CorrectnessPtr(nContent)\uparrow.nJustification,nBlock);\ Print\_String(\ ;\ );
  Print_NewLine;
  end:
itProperty: \mathbf{begin} \ Print\_String(PropertyName[PropertyPtr(nContent)\uparrow.nPropertySort]);
  Print\_Justification(PropertyPtr(nContent)\uparrow.nJustification,nBlock);\ Print\_String(`;`);
  Print_NewLine;
  end;
itDefMode: with ModeDefinitionPtr(nContent) \uparrow do
    begin if nRedefinition then
      begin Print_String(TokenName[sy_Redefine]);
      end;
    Print_String(TokenName[sy_Mode]); Print_Pattern(nDefModePattern);
    case nDefKind of
    defExpandableMode: begin Print_String(TokenName[sy_Is]);
      Print_Type(ExpandableModeDefinitionPtr(nContent)\uparrow.nExpansion);
      end:
    defStandardMode: with StandardModeDefinitionPtr(nContent) \uparrow do
         begin if nSpecification \neq nil then
           begin Print_String(TokenName[sy_Arrow]); Print_Type(nSpecification);
           end:
         if nDefiniens \neq nil then
           begin Print_String(TokenName[sy_Means]); Print_NewLine; Print_Definiens(nDefiniens);
           end;
         end;
    endcases; Print_String(';'); Print_NewLine;
itDefAttr: with AttributeDefinitionPtr(nContent) \uparrow do
    begin if nRedefinition then
      begin Print_String(TokenName[sy_Redefine]);
      end;
    Print\_String(TokenName[sy\_Attr]); Print\_Pattern(nDefAttrPattern);
    Print_String(TokenName[sy_Means]); Print_NewLine; Print_Definiens(nDefiniens);
    Print_String(`;`); Print_NewLine;
    end:
itDefPred: with PredicateDefinitionPtr(nContent) \uparrow do
    begin if nRedefinition then
      begin Print_String(TokenName[sy_Redefine]);
    Print_String(TokenName[sy_Pred]); Print_Pattern(nDefPredPattern);
    if nDefiniens \neq nil then
      begin Print_String(TokenName[sy_Means]); Print_NewLine; Print_Definiens(nDefiniens);
```

```
Print_String(';'); Print_NewLine;
    end:
itDefFunc: with FunctorDefinitionPtr(nContent) \uparrow do
    begin if nRedefinition then
       begin Print_String(TokenName[sy_Redefine]);
       end;
    Print_String(TokenName[sy_Func]); Print_Pattern(nDefFuncPattern);
    if nSpecification \neq nil then
       \textbf{begin} \ \textit{Print\_String}(\textit{TokenName}[\textit{sy\_Arrow}]); \ \textit{Print\_Type}(\textit{nSpecification});
       end:
    case nDefiningWay of
    dfEmpty:
    dfMeans: begin Print_String(TokenName[sy_Means]); Print_NewLine;
    dfEquals: begin Print_String(TokenName[sy_Equals]);
    endcases; Print_Definiens(nDefiniens); Print_String(`;`); Print_NewLine;
itDefStruct: with StructureDefinitionPtr(nContent)↑ do
    begin Print_String(TokenName[sy_Struct]);
    if nAncestors \uparrow. Count > 0 then
       begin Print\_String(`(`); Print\_Type(nAncestors \uparrow .Items \uparrow [0]);
       for i \leftarrow 1 to nAncestors \uparrow. Count - 1 do
         begin Print\_String(`,`); Print\_Type(nAncestors\uparrow.Items\uparrow[i]);
         end:
       Print_String(`)`);
       end:
    Print\_String(StructureName[nDefStructPattern \uparrow. nModeSymbol]);
    if (nDefStructPattern\uparrow.nArgs \neq nil) \land (nDefStructPattern\uparrow.nArgs\uparrow.Count > 0) then
       begin Print\_String(TokenName[sy\_Over]); Print\_Loci(nDefStructPattern \uparrow .nArgs);
    Print_String(TokenName[sy_StructLeftBracket]);
    for i \leftarrow 0 to nSqmFields \uparrow. Count - 1 do
       with FieldSegmentPtr(nSgmFields\uparrow.Items\uparrow[i])\uparrow do
         begin Print\_String(SelectorName[FieldSymbolPtr(nFields\\uparrow.Items\\uparrow[0])\\uparrow.nFieldSymbol]);
         for j \leftarrow 1 to nFields \uparrow. Count - 1 do
            with FieldSymbolPtr(nFields\uparrow.Items\uparrow[j])\uparrow do
              begin Print_String(`,`); Print_String(SelectorName[nFieldSymbol]);
              end:
         Print\_String(TokenName[sy\_Arrow]); Print\_Type(nSpecification);
         if (i \ge 0) \land (i < nSgmFields \uparrow. Count - 1) then Print\_String(`, `);
         end:
    Print_String(TokenName[sy_StructRightBracket]); Print_String(';'); Print_NewLine;
    end:
itPredSynonym, itFuncNotation, itModeNotation, itAttrSynonym:
         with NotationDeclarationPtr(nContent) \uparrow do
    begin Print_String(TokenName[sy_Synonym]); Print_Pattern(nNewPattern);
    Print_String(TokenName[sy_For]); Print_Pattern(nOriginPattern); Print_String(';');
    Print_NewLine;
    end;
itPredAntonym, itAttrAntonym: with NotationDeclarationPtr(nContent) \uparrow do
    begin Print_String(TokenName[sy_Antonym]); Print_Pattern(nNewPattern);
```

```
Print_String(TokenName[sy_For]); Print_Pattern(nOriginPattern); Print_String(';');
    Print_NewLine;
    end:
itCluster: begin Print_String(TokenName[sy_Cluster]);
  case ClusterPtr(nContent)\uparrow .nClusterKind of
  ExistentialRegistration: with EClusterPtr(nContent) \uparrow do
      begin Print_AdjectiveList(nConsequent); Print_String(TokenName[sy_For]);
      Print_Type(nClusterType);
      end:
  ConditionalRegistration: with CClusterPtr(nContent) \uparrow do
      begin Print_AdjectiveList(nAntecedent); Print_String(TokenName[sy_Arrow]);
      Print_AdjectiveList(nConsequent); Print_String(TokenName[sy_For]);
      Print_Type(nClusterType);
      end:
  FunctorialRegistration: with FClusterPtr(nContent) \uparrow do
      begin Print_Term(nClusterTerm); Print_String(TokenName[sy_Arrow]);
      Print\_AdjectiveList(nConsequent);
      if nClusterType \neq nil then
         begin Print_String(TokenName[sy_For]); Print_Type(nClusterType);
         end:
      end;
  endcases; Print_String(';'); Print_NewLine;
itIdentify: with IdentifyRegistrationPtr(nContent) \uparrow do
    begin Print_String(TokenName[sy_Identify]); Print_Pattern(nNewPattern);
    Print\_String(TokenName[sy\_With]); Print\_Pattern(nOriginPattern);
    if (nEqLociList \neq nil) \land (nEqLociList \uparrow. Count > 0) then
      begin Print_String(TokenName[sy_When]);
      for i \leftarrow 0 to nEqLociList \uparrow. Count - 1 do
         with LociEqualityPtr(nEqLociList\uparrow.Items\uparrow[i])\uparrow do
           begin Print_Locus(nLeftLocus); Print_String('='); Print_Locus(nRightLocus);
           if (i \ge 0) \land (i < nEqLociList \uparrow. Count - 1) then Print\_String( `, `);
           end;
      end;
    Print_String(';'); Print_NewLine;
itPropertyRegistration: case PropertyRegistrationPtr(nContent) \uparrow .nPropertySort of
  sySethood: with SethoodRegistrationPtr(nContent) \uparrow do
      begin Print_String(PropertyName[nPropertySort]); Print_String(TokenName[sy_Of]);
       Print\_Type(nSethoodType); Print\_Justification(nJustification, nBlock); Print\_String(`;`);
       Print_NewLine;
      end:
  endcases:
itReduction: begin with ReduceRegistrationPtr(nContent) \uparrow do
    begin Print_String(TokenName[sy_Reduce]); Print_Term(nOriginTerm);
    Print\_String(TokenName[sy\_To]); Print\_Term(nNewTerm);
    end;
  Print_String(`;`); Print_NewLine;
itPragma: \mathbf{begin} \ Print\_NewLine; \ Print\_String(`::` + PragmaPtr(nContent) \uparrow .nPragmaStr);
  Print_NewLine;
  end;
```

```
itIncorrItem: ; \\ end; \\ endcases; \\ end; \\ \\ 1192. \quad \langle \text{Implementation for wsmarticle.pas } 854 \rangle + \equiv \\ \\ \text{procedure } Print\_WSMizArticle(aWSTextProper: wsTextProperPtr; aFileName: string); \\ \\ \text{var } lWSMizOutput: WSMizarPrinterPtr; \\ \\ \text{begin } InitScannerNames; lWSMizOutput \leftarrow new(WSMizarPrinterPtr, OpenFile(aFileName)); \\ lWSMizOutput \uparrow .Print\_TextProper(aWSTextProper); dispose(lWSMizOutput, Done); \\ \\ \text{end}; \\ \end{aligned}
```

420 DETOUR: PRAGMAS Mizar Parser §1193

File 22

Detour: Pragmas

1193. This chapter is a "detour" because it is out of order for the compiler, but it is a dependency for the next file (parseradditions.pas).

The base/pragmas.pas contains the global variables which are toggled by pragmas like "::\$P+". This will toggle the *ProofPragma*. In particular, when *ProofPragma* is true, then Mizar will double check the proofs. When *ProofPragma* is false, Mizar will skip the proofs.

```
⟨ gnu License 4⟩
unit pragmas;
interface uses mobjects;
var VerifyPragmaOn, VerifyPragmaOff: NatSet; VerifyPragmaIntervals: NatFunc;
    SchemePragmaOn, SchemePragmaOff: NatSet; SchemePragmaIntervals: NatFunc;
    ProofPragma: Boolean = true; { check the proofs? }

procedure SetParserPragma(aPrg: string);
procedure InsertPragma(aLine: integer; aPrg: string);
procedure CompletePragmas(aLine: integer);
procedure CanceledPragma ( const aPrg: string; var aKind: char; var aNbr: integer );
implementation
uses mizenv;
```

1194. Cancelling a definition or theorem is handled with the "::\$C" pragma, which is administered only by the editors of the MML.

```
procedure CanceledPragma ( const aPrg: string; var aKind: char; var aNbr: integer ); var lStr: string; k, lCod: integer; begin aKind \leftarrow ``\_`; if (Copy(aPrg, 1, 2) = `$C`) then begin if (length(aPrg) \geq 3) \wedge (aPrg[3] \in [`D`, `S`, `T`]) then begin aKind \leftarrow aPrg[3]; lStr \leftarrow TrimString(Copy(aPrg, 4, length(aPrg) - 3)); aNbr \leftarrow 1; if length(lStr) > 0 then begin k \leftarrow 1; while (k \leq length(lStr)) \wedge (lStr[k] \in [`0`...`9`]) do inc(k); delete(lStr, k, length(lStr)); if length(lStr) > 0 then Val(lStr, aNbr, lCod); end; end; end;
```

 $\S1195$ Mizar Parser DETOUR: PRAGMAS 421

```
The "::$P+" pragma instructs Mizar to start checking the proofs for correctness. The "::$P-"
pragma instructs Mizar to skip checking proofs.
procedure SetParserPragma(aPrg:string);
  begin if copy(aPrq, 1, 3) = \text{`$P+'$ then}
    begin ProofPragma \leftarrow true;
    end:
  if copy(aPrg, 1, 3) =  '$P-' then
    begin ProofPragma \leftarrow false;
    end;
  end:
1196. The "::$S+" pragma will tell Mizar to check the scheme references, whereas "::$S-" pragma tells
Mizar to stop verifying scheme references.
  The "::$V+" pragma enables the verifier, and the "::$V-" pragma disables the verifier (skipping all
verification until it is re-enabled).
procedure InsertPragma(aLine: integer; aPrg: string);
  begin if copy(aPrg, 1, 3) = \text{`$V+'$ then}
    begin VerifyPragmaOn.InsertElem(aLine); end;
  if copy(aPrq, 1, 3) = \text{`$V-'} then
    begin VerifyPragmaOff.InsertElem(aLine); end;
  if copy(aPrq, 1, 3) =  $S+\cdot then
    begin SchemePragmaOn.InsertElem(aLine); end;
  if copy(aPrg, 1, 3) =  $S-\tag{then}
    begin SchemePragmaOff.InsertElem(aLine); end;
  end:
1197. The Complete Pragmas function will compute the intervals for which the pragmas are "active", then
check whether the given line number falls within the "active range".
procedure CompletePragmas(aLine : integer);
  var i, j, a, b: integer; f: boolean;
  begin for i \leftarrow 0 to VerifyPragmaOff.Count - 1 do
    begin f \leftarrow false; \ a \leftarrow VerifyPragmaOff.Items \uparrow [i].X;
    for j \leftarrow 0 to VerifyPragmaOn.Count - 1 do
       begin b \leftarrow VerifyPragmaOn.Items\uparrow[j].X;
      if b \ge a then
         begin VerifyPragmaIntervals.Assign(a, b); f \leftarrow true; break; end;
    if \neg f then VerifyPragmaIntervals.Assign(a, aLine);
  for i \leftarrow 0 to SchemePragmaOff.Count - 1 do
    begin f \leftarrow false; \ a \leftarrow SchemePragmaOff.Items \uparrow [i].X;
    for j \leftarrow 0 to SchemePragmaOn.Count - 1 do
```

begin $b \leftarrow SchemePragmaOn.Items\uparrow[j].X;$

if $\neg f$ then SchemePragmaIntervals.Assign(a, aLine);

begin $SchemePragmaIntervals.Assign(a,b); f \leftarrow true; break; end;$

if $b \ge a$ then

end; end; 422 DETOUR: PRAGMAS Mizar Parser $\S1198$

1198. Now we initialize the global variables declared in this module.

 $\label{eq:begin VerifyPragmaOn.Init} \begin VerifyPragmaOn.Init(10,10); VerifyPragmaIntervals.InitNatFunc(10,10); SchemePragmaOn.Init(10,10); SchemePragmaIntervals.InitNatFunc(10,10); end.$

Detour: Parser additions

1199. This chapter is a "detour" because we are "going out of [compiler] order" to discuss parseradditions.pas. Why? Well, because the file provides subclasses to those introduced in the abstract syntax unit, and are necessary for understanding the parser.pas unit.

One of the difficulties with this file is that there are 37 global variables declared here, and 46 module-wide variables, declared here. It's hard to juggle that knowledge! These "global" variables really describe the state of the parser, and do not seem to be used anywhere else.

[It would probably be wise to refactor the design to isolate these variables inside a Parser class, so they are not randomly distributed throughout this part of the program.]

CONVENTIONS: The classes have methods prefixed by Start, Process, and Finish.

- The Start methods reset the state variables needed to parse the syntactic entity.
- The *Process* methods usually update the state variables, either allocating new objects or transferring the current contents of a state variable in a different state variable.
- The Finish methods construct a WSM abstract syntax tree for the parsed entity.

```
\langle \text{ parseraddition.pas } 1199 \rangle \equiv
  ⟨GNU License 4⟩
unit parseraddition;
  interface
  uses syntax, errhan, mobjects, mscanner, abstract_syntax, wsmarticle, xml_inout;
  procedure InitWsMizarArticle;
  type
    (Extended block class declaration 1205)
     Extended item class declaration 1225
     Extended subexpression class declaration 1388
    (Extended expression class declaration 1489)
  function GetIdentifier: integer;
  function CreateArgs(aBase: integer): PList;
  var (Global variables introduced in parseraddition.pas 1202)
  implementation
  uses mizenv, mconsole, parser, _formats, pragmas
      mdebug , info end_mdebug;
  const MaxSubTermNbr = 64;
  var (Local variables for parser additions 1209)
    (Implementation of parser additions 1200)
  end.
```

Mizar Parser

```
424
1200.
        \langle Implementation of parser additions 1200 \rangle \equiv
  (Get the identifier number for current word 1201)
  (Initialize WS Mizar article 1203);
   Extended block implementation 1206
   Extended item implementation 1226 >
   (Extended subexpression implementation 1390)
  (Extended expression implementation 1490)
This code is used in section 1199.
identifier, we should return 0.
```

1201. When the current token is an identifier, we should obtain its number. If the current token is not an

```
\langle Get the identifier number for current word 1201\rangle \equiv
function GetIdentifier: integer;
  begin result \leftarrow 0;
  if CurWord.Kind = Identifier then result \leftarrow CurWord.Nr
  end:
This code is used in section 1200.
```

1202. Initializing a weakly-strict Mizar article requires setting the values for some of the global variables. Importantly, this will initialize the qBlockPtr in the Parser to be an extBlockObj instance. Note that this will create "the" blMain block object.

```
\langle Global variables introduced in parseraddition.pas 1202 \rangle \equiv
gWSTextProper: wsTextProperPtr;
gLastWSBlock: WSBlockPtr;
gLastWSItem: WSItemPtr;
See also sections 1210, 1212, 1227, 1231, 1234, 1240, 1243, 1247, 1256, 1268, 1273, 1287, 1297, 1308, 1316, 1320, 1328, 1336,
     1338, 1340, 1346, 1348, 1367, 1370, 1374, and 1394.
This code is used in section 1199.
1203. \langle Initialize WS Mizar article 1203\rangle \equiv
procedure InitWsMizarArticle;
             { inintialize global variables which were declared in parseraddition }
  qWSTextProper \leftarrow new(wsTextProperPtr, Init(ArticleID, ArticleExt, CurPos));
  gLastWSBlock \leftarrow gWSTextProper; gLastWSItem \leftarrow \mathbf{nil};
  qBlockPtr \leftarrow new(extBlockPtr, Init(blMain)); {initialize other global variables}
  end;
```

This code is used in section 1200.

 $\{1204$ Mizar Parser EXTENDED BLOCK CLASS 425

Section 23.1. EXTENDED BLOCK CLASS

1204. We extend the *Block* class ($\S691$) introduced in the syntax.pas unit. Also recall the *wsBlock* class ($\S856$) and the *wsItem* class ($\S860$).

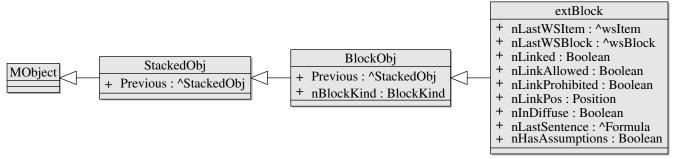


Fig. 6. Class hierarchy for extBlockObj, methods omitted.

```
\langle Extended block class declaration 1205\rangle \equiv
1205.
  extBlockPtr = \uparrow extBlockObj;
  extBlockObj = \mathbf{object} (BlockObj)
    nLastWSItem: WSItemPtr;
    nLastWSBlock: WSBlockPtr;
    nLinked: Boolean; { is block prefixed by "then"? }
    nLinkAllowed: Boolean; { isn't this a duplicate of next field? }
    nLinkProhibited: Boolean; { can statement kind be prefixed by "then"? }
    nLinkPos: Position;
    nInDiffuse: boolean:
    nLastSentence: FormulaPtr;
    nHasAssumptions: Boolean;
    constructor Init(fBlockKind : BlockKind);
    procedure Pop; virtual;
    procedure StartProperText; virtual;
    procedure ProcessRedefine; virtual;
    procedure ProcessLink; virtual;
    procedure ProcessBegin; virtual;
    procedure ProcessPragma; virtual;
    procedure StartSchemeDemonstration; virtual;
    procedure FinishSchemeDemonstration; virtual;
    procedure CreateItem(fItemKind : ItemKind); virtual;
    procedure CreateBlock(fBlockKind : BlockKind); virtual;
  end;
```

This code is used in section 1199.

1206. Constructor. The constructor for an extended block object invokes the parent class's constructor (§694), initializes the instance variables, then its behaviour depends on whether we are constructing a "main" block or not.

```
\langle Extended block implementation 1206\rangle \equiv
constructor extBlockObj.Init(fBlockKind : BlockKind);
  begin inherited Init (fBlockKind);
  \langle Initialize default values for extBlock instance 1207\rangle;
  if nBlockKind = blMain then \langle Initialize main extBlock instance 1208 \rangle
  else (Initialize "proper text" extBlock instance 1211);
See also sections 1213, 1214, 1215, 1216, 1217, 1218, 1219, 1221, 1222, and 1223.
This code is used in section 1200.
1207. We have the default values suppose links are prohibited for the block, and there are no assumptions
for the block. The last wsItem and wsBlock pointers are set to the global gLastWSItem and gLastWSBlock
variables, respectively.
\langle \text{Initialize default values for } extBlock \text{ instance } 1207 \rangle \equiv
  nLinked \leftarrow false; nLinkPos \leftarrow CurPos; nLinkAllowed \leftarrow false; nLinkProhibited \leftarrow true;
  nHasAssumptions \leftarrow false; gRedefinitions \leftarrow false;
  nLastWSItem \leftarrow gLastWSItem; nLastWSBlock \leftarrow gLastWSBlock;
This code is used in section 1206.
         The "main" block of text needs to load the formats file, and populate the gFormatsColl (§648) and
the gFormatsBase (ibid.) global variables. The parseraddition.pas unit's gProofCnt global variable is
initialized to zero here.
\langle \text{Initialize main } extBlock \text{ instance } 1208 \rangle \equiv
  begin nInDiffuse \leftarrow true; gProofCnt \leftarrow 0;
  FileExam(EnvFileName + `.frm'); \ gFormatsColl.LoadFormats(EnvFileName + `.frm');
  gFormatsBase \leftarrow gFormatsColl.Count; setlength(Term, MaxSubTermNbr);
  end
This code is used in section 1206.
1209. \langle Local variables for parser additions 1209 \rangle \equiv
Term: array of TermPtr; \{(\S753)\}
See also sections 1228, 1232, 1238, 1241, 1244, 1245, 1248, 1252, 1258, 1260, 1262, 1269, 1271, 1274, 1278, 1284, 1292, 1298,
     1302, 1309, 1317, 1331, 1382, and 1389.
This code is used in section 1199.
```

 \langle Global variables introduced in parseraddition.pas $1202 \rangle + \equiv$

gProofCnt: integer;

§1211 Mizar Parser EXTENDED BLOCK CLASS 427

1211. The "proper text" branch updates the gLastWSBlock global variable. For most of the kinds of blocks, we will have to toggle nInDiffuse to be true or false. For proof blocks, we will need to increment the "depth" counter tracking the proof block "nestedness".

Only the "case" and "suppose" blocks, when determining if they are in "diffuse mode" or not, need to confer with the previous block. (Recall (§307), StackedObj classes has a Previous pointer.)

```
\langle \text{Initialize "proper text" } extBlock \text{ instance } 1211 \rangle \equiv
  begin gLastWSBlock \leftarrow gWsTextProper \uparrow.NewBlock(nBlockKind, CurPos);
  mizassert(2341, gLastWSItem \neq nil);
  if qLastWSItem \uparrow .nItemKind \in [itDefinition, itRegularStatement, itSchemeBlock, itTheorem,
          itConclusion, itCaseBlock, itCorrCond, itCorrectness, itProperty, itPropertyRegistration then
     wsItemPtr(gLastWSItem).nBlock \leftarrow gLastWSBlock;
  case nBlockKind of
  blDefinition: nInDiffuse \leftarrow false;
  blNotation: nInDiffuse \leftarrow false;
  blDiffuse: nInDiffuse \leftarrow true;
  blHereby: nInDiffuse \leftarrow true;
  blProof: \mathbf{begin} \ nLastSentence \leftarrow gLastFormula; \ inc(gProofCnt); \mathbf{end};
  blCase: nInDiffuse \leftarrow extBlockPtr(Previous) \uparrow .nInDiffuse;
  blSuppose: nInDiffuse \leftarrow extBlockPtr(Previous) \uparrow .nInDiffuse;
  blRegistration: nInDiffuse \leftarrow false;
  blPublicScheme: nInDiffuse \leftarrow false;
  endcases:
  end
```

This code is used in section 1206.

1212. Popping a block. When we "pop" a proof block, we need to track the formula that was just proven and store it in the global variable *qLastFormula*.

```
\langle Global variables introduced in parseraddition.pas 1202 \rangle += gLastFormula: FormulaPtr;
```

1213. This actually implements the Pop method for blocks. When a block "closes" (i.e., the corresponding "end" statement has been encountered), we restore the global state's gLastWSItem and gLastWSBlock pointers. When a proof block closes, we also restore the gLastFormula state.

Also note: the parent class's method (§695) does nothing. This will be invoked in the KillBlock (§687).

```
\langle Extended block implementation 1206\rangle +\equiv procedure extBlockObj.Pop; begin gLastWSBlock\uparrow.nBlockEndPos \leftarrow CurPos; case nBlockKind of blProof: begin gLastFormula \leftarrow nLastSentence; dec(gProofCnt); end; endcases; gLastWSItem \leftarrow nLastWSItem; gLastWSBlock \leftarrow nLastWSBlock; { restore the "last" pointers } inherited Pop; end;
```

Mizar Parser §1214

1214. Process "begin". Mizar uses "begin" to start a new "section" at the top-level of an article. Recall the grammar for this bit of Mizar:

```
Text-Proper = Section { Section } .
Section = "begin" { Text-Item } .
```

There are zero or more Text-Items in a section.

We should note that the main text is not organized as a linked list of "main" blocks. Instead, we have a single "main" block, and we just push an *itSection* item to its contents.

```
\langle Extended block implementation 1206\rangle += procedure extBlockObj.ProcessBegin; begin nLinkAllowed \leftarrow false; nLinkProhibited \leftarrow true; gLastWSItem \leftarrow gWsTextProper\uparrow.NewItem(itSection, CurPos); nLastWSItem \leftarrow gLastWSItem; gLastWSBlock\uparrow.nItems.Insert(gLastWSItem); end;
```

1215. This will add a pragma item to the current block. The parser's *ProcessPragmas* (§1585) invokes this method.

```
⟨ Extended block implementation 1206⟩ +≡ procedure extBlockObj.ProcessPragma;
begin nLinkAllowed \leftarrow false; nLinkProhibited \leftarrow true;
{ Create a new item }
gLastWSItem \leftarrow gWsTextProper \uparrow.NewItem(itPragma, CurPos);
gLastWSItem \uparrow.nContent \leftarrow new(PragmaPtr, Init(CurWord.Spelling));
{ Insert the pragma, update last item in block }
nLastWSItem \leftarrow gLastWSItem; gLastWSBlock \uparrow.nItems.Insert(gLastWSItem);
end;
```

1216. Starting the proper text will just update the nBlockPos field to whatever the current position is.

```
⟨ Extended block implementation 1206⟩ +≡
procedure extBlockObj.StartProperText;
begin gWSTextProper↑.nBlockPos ← CurPos; end;
```

1217. Processing redefinitions sets the global variable gRedefinitions to the result of comparing the current word to the "redefine" keyword.

```
⟨ Extended block implementation 1206⟩ +≡
procedure extBlockObj.ProcessRedefine;
begin gRedefinitions ← CurWord.Kind = sy_Redefine; end;
```

1218. When a block statement is linked, but it should not, then we raise a 164 error. Otherwise, be sure to mark the block as linked (i.e., toggle nLinked to be true) and assign the nLinkPos to be the current position.

```
\langle Extended block implementation 1206\rangle += procedure extBlockObj.ProcessLink; begin if CurWord.Kind \in [sy\_Then, sy\_Hence] then begin if nLinkProhibited then ErrImm(164); nLinked \leftarrow true; nLinkPos \leftarrow CurPos; end; end;
```

§1219 Mizar Parser EXTENDED BLOCK CLASS 429

1219. Proof of a scheme. We should increment the proof depth global variable.

```
Recall that ProofPragma means "check the proof is valid?" In other words, when ProofPragma is false, we are skipping the proofs.
```

```
§thesis-formula:macro-def

define thesis\_formula \equiv new(ThesisFormulaPtr, Init(CurPos))

define thesis\_prop \equiv new(PropositionPtr, Init(new(LabelPtr, Init(0, CurPos)), thesis\_formula, CurPos))

define skipped\_proof\_justification \equiv new(JustificationPtr, Init(infSkippedProof, CurPos))

\langle \text{Extended block implementation 1206} \rangle + \equiv

procedure extBlockObj.StartSchemeDemonstration;

begin inc(gProofCnt);

if \neg ProofPragma then \langle \text{Mark schema proof as "skipped" 1220} \rangle;

end;
```

1220. When we skip the proof (due to pragmas being set), we just add the scheme as a compact statement whose justification is the "skipped proof justification".

First, we create a new text item for the proper text global variable. Then we set its content to the compact statement with the "skipped" justification. Finally we add this item to the "last" (latest) wsBlock global variable.

```
\langle \text{Mark schema proof as "skipped" } 1220 \rangle \equiv  begin gLastWSItem \leftarrow gWsTextProper \uparrow.NewItem(itConclusion, CurPos); <math>gLastWSItem \uparrow.nContent \leftarrow new(CompactStatementPtr, Init(thesis\_prop, skipped\_proof\_justification)); <math>gLastWSBlock \uparrow.nItems.Insert(gLastWSItem); end
```

This code is used in section 1219.

1221. Finishing the proof for a scheme should decrement the global "proof depth" counter.

```
⟨ Extended block implementation 1206⟩ +≡
procedure extBlockObj.FinishSchemeDemonstration;
begin dec(gProofCnt); end;
```

1222. The factory method for extBlock creating an item will update the global gItemPtr variable (§690). \langle Extended block implementation $1206\rangle +\equiv$

```
procedure extBlockObj.CreateItem(fItemKind : ItemKind); begin gItemPtr \leftarrow new(extItemPtr, Init(fItemKind)); end;
```

1223. The factory method for extBlock creating a new block will update the gBlockPtr global variable ($\S690$).

```
\langle Extended block implementation 1206\rangle +\equiv procedure extBlockObj.CreateBlock(fBlockKind: BlockKind); begin <math>gBlockPtr \leftarrow new(extBlockPtr,Init(fBlockKind)) end;
```

430 EXTENDED ITEM CLASS Mizar Parser §1224

Section 23.2. EXTENDED ITEM CLASS

1224. The class diagram for extended items looks like:

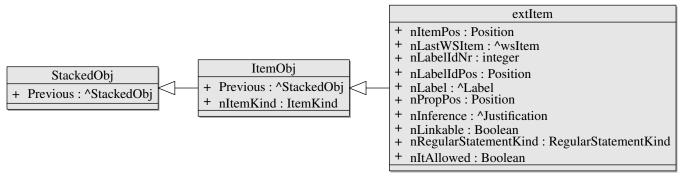


Fig. 7. Class hierarchy for extItemObj. The base MObject class omitted from the hierarchy.

Recall (§909) the regular statement kind is one of three possibilities: diffuse statement, compact statement, iterative equality.

The "Finish" methods updates the contents of the *extItem* class with a WSM abstract syntax tree for the statement.

Since this is a "stub", I will just leave the placeholder chunk for the methods overriden by the extended Item class here (remove later).

 \langle Methods overriden by extended Item class 706 $\rangle + \equiv$

```
\langle Extended item class declaration 1225\rangle \equiv
  extItemPtr = \uparrow extItemObj;
  extItemObj = \mathbf{object} (ItemObj)
    nItemPos: Position;
    nLastWSItem: WSItemPtr;
    nLabelIdNr: integer;
    nLabelIdPos: Position;
    nLabel: LabelPtr;
    nPropPos: Position;
    nInference: JustificationPtr;
    nLinkable: boolean;
    nRegularStatementKind: RegularStatementKind;
    nItAllowed: boolean;
    constructor Init(fKind : ItemKind);
    procedure Pop; virtual;
    (Methods overriden by extended Item class 706)
  end;
This code is used in section 1199.
```

§1226 Mizar Parser CONSTRUCTOR 431

Subsection 23.2.1. Constructor

1226. There are a number of comments in Polish which I haphazardly translated into English ("Przygotowanie definiensow:" translates as "Preparation of definiens:"; "Ew. zakaz przy obiektach ekspandowanych" translates as "Possible ban on expanded facilities")

```
\langle Extended item implementation 1226\rangle \equiv
constructor extItemObj.Init(fKind : ItemKind);
      begin inherited Init (fKind):
      (Initialize the fields for newly allocated extItem object 1229)
      mizassert(2343, gLastWSBlock \neq nil);
      if \neg (nItemKind \in [itReservation, itConstantDefinition, itExemplification, itGeneralization,
                         itLociDeclaration]) then
            begin qLastWSItem \leftarrow qWsTextProper\uparrow.NewItem(fKind, CurPos); nLastWSItem \leftarrow qLastWSItem;
            end:
      case nItemKind of
            (Initialize extended item by ItemKind 1230)
      endcases;
      if \neg (nItemKind \in [itReservation, itConstantDefinition, itExemplification, itGeneralization,
                         itLociDeclaration]) then qLastWSBlock \uparrow .nItems.Insert(qLastWSItem);
      end;
See also sections 1249, 1277, 1279, 1280, 1281, 1282, 1283, 1285, 1286, 1288, 1289, 1290, 1291, 1293, 1294, 1295, 1296, 1299,
            1300, 1301, 1303, 1304, 1305, 1306, 1307, 1310, 1311, 1312, 1313, 1314, 1315, 1318, 1319, 1321, 1322, 1323, 1324, 1325,
            1326, 1327, 1329, 1330, 1332, 1333, 1334, 1335, 1337, 1339, 1341, 1342, 1343, 1344, 1345, 1347, 1349, 1350, 1351, 1352,
            1353,\ 1354,\ 1355,\ 1356,\ 1357,\ 1358,\ 1359,\ 1360,\ 1361,\ 1362,\ 1363,\ 1364,\ 1365,\ 1366,\ 1368,\ 1369,\ 1371,\ 1372,\ 1373,\ 1375,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 1376,\ 
            1376, 1377, 1378, 1379, 1380, 1381, 1383, 1384, and 1385.
This code is used in section 1200.
```

1227. Initializing the fields. The *it_Allowed* global variable is toggled on and off when the parser encounters "guards" in conditional definitions, whereas the *nItAllowed* fields reflects whether the sort of definition allows "it" in the definiens.

```
\langle Global variables introduced in parseraddition.pas 1202\rangle +\equiv dol_Allowed: Boolean; it_Allowed: Boolean; in_AggrPattern: Boolean; gLastType: TypePtr; gLastTerm: TermPtr; gDefiningWay: HowToDefine; 

1228. \langle Local variables for parser additions 1209\rangle +\equiv gClusterSort: ClusterRegistrationKind; gDefiniens: DefiniensPtr; gPartialDefs: PList; nDefiniensProhibited: boolean; qSpecification: TypePtr;
```

432 CONSTRUCTOR Mizar Parser §1229

```
1229.
         (Initialize the fields for newly allocated extItem object 1229)
  nItemPos \leftarrow CurPos; qClusterSort \leftarrow ExistentialRegistration; nItAllowed \leftarrow false; it\_Allowed \leftarrow false;
       { global variable! }
  in\_AggrPattern \leftarrow false; \ dol\_Allowed \leftarrow false; \ gSpecification \leftarrow \mathbf{nil}; \ gLastType \leftarrow \mathbf{nil};
  gLastFormula \leftarrow \mathbf{nil}; \ gLastTerm \leftarrow \mathbf{nil};
     { Preparation of definiens: }
  nDefiniensProhibited \leftarrow false;
     { Possible ban on expanded facilities }
  gDefiningWay \leftarrow dfEmpty; \ gDefiniens \leftarrow \mathbf{nil}; \ gPartialDefs \leftarrow \mathbf{nil}; \ nLinkable \leftarrow false;
This code is used in section 1226.
1230. Kind-specific initialization. Each kind of item may need some specific initialization. We work
through all the cases. The first two cases considered are generalization ("let \( Qualified Variables \) be [such
\langle Conditions \rangle]") and existential assumptions ("given \langle Qualified\ Variables \rangle such \langle Conditions \rangle"). Existential
assumptions need to toggle the "has assumptions" field to true for the global block pointer.
\langle \text{Initialize extended item by } ItemKind | 1230 \rangle \equiv
itGeneralization: ; {let statements}
itExistentialAssumption: ExtBlockPtr(gBlockPtr) \uparrow .nHasAssumptions \leftarrow true;
See also sections 1233, 1235, 1237, 1239, 1242, and 1246.
This code is used in sections 1226 and 1244.
1231. Property initialization. Initializing a property statement Item should raise an error when the
property does not appear in the correct block.
• Defining a predicate can support the following properties: symmetry, reflectivity, irreflexivity, transitivity,
conectedness, asymmetry.
• Functors can support: associativity, commutativity, idempotence, involutiveness, and projectivity proper-
• Modes can support the sethood property.
  In all other situations, an error should be flagged (the user is trying to assert an invalid property).
\langle Global variables introduced in parseraddition.pas 1202\rangle + \equiv
gDefKind: ItemKind;
1232. \langle \text{Local variables for parser additions 1209} \rangle + \equiv
qExpandable: boolean;
gPropertySort: PropertyKind;
1233. (Initialize extended item by ItemKind 1230) +\equiv
itProperty: \mathbf{begin} \ qPropertySort \leftarrow PropertyKind(CurWord.Nr);
  case PropertyKind(CurWord.Nr) of
  sySymmetry, syReflexivity, syIrreflexivity, syTransitivity, syConnectedness, syAsymmetry:
     if gDefKind \neq itDefPred then
       begin ErrImm(81); gPropertySort \leftarrow sErrProperty; end;
  syAssociativity, syCommutativity, syIdempotence: if gDefKind \neq itDefFunc then
       begin ErrImm(82); gPropertySort \leftarrow sErrProperty; end;
  syInvolutiveness, syProjectivity: if gDefKind \neq itDefFunc then
       begin ErrImm(83); gPropertySort \leftarrow sErrProperty; end;
  sySethood: if (gDefKind \neq itDefMode) \lor gExpandable then
       begin ErrImm(86); gPropertySort \leftarrow sErrProperty; end;
  endcases;
  end;
```

§1234 Mizar Parser CONSTRUCTOR 433

1234. Reconsider initialization. We need to allocate a new (empty) list for the list of terms being reconsidered.

```
\langle Global variables introduced in parseraddition.pas 1202 \rangle += gReconsiderList\colon PList;
```

```
1235. (Initialize extended item by ItemKind\ 1230) +\equiv itReconsider: gReconsiderList \leftarrow new(PList, Init(0));
```

1236. We can have in Mizar "suppose that $\langle statement \rangle$ " (as well as "case that..."). But in those cases, the statement cannot be linked to the next statement (i.e., the next statement cannot begin with "then..."). Assumptions without "that" are always linkable.r

Theorems, "regular statements", and conclusions are always linkable.

```
1237. \langle \text{Initialize extended item by } \textit{ItemKind } 1230 \rangle +\equiv itRegularStatement: nLinkable \leftarrow true; itConclusion: nLinkable \leftarrow true; itPerCases: ; itPerCases: ; itCaseHead: if AheadWord.Kind <math>\neq sy_That then nLinkable \leftarrow true; itSupposeHead: if AheadWord.Kind \neq sy_That then nLinkable \leftarrow true; itTheorem: nLinkable \leftarrow true; itAxiom: if \neg AxiomsAllowed then ErrImm(66); itChoice: ;
```

1238. Initializing an assumption. Collective assumptions ("assume that $\langle formula \rangle$ ") are not linkable, but single assumptions ("assume $\langle Proposition \rangle$ ") are linkable. The statement will introduce a list of premises, which will be tracked in the qPremises local variable for the module.

```
gPremises: PList; 

1239. \langle Initialize extended item by ItemKind\ 1230 \rangle + \equiv itAssumption: begin if AheadWord.Kind \neq sy\_That\ then\ nLinkable \leftarrow true;
```

 $\langle \text{Local variables for parser additions } 1209 \rangle + \equiv$

gDefPos: Position; gStructPrefixes: PList;

Assumption: begin if AheadWord.Kind \neq sy_That then nLinkable \notin gPremises \leftarrow nil; end;

1240. Definition items. Definition items need to be initialized with some nuance. Some definitions permit "it" to be used in the definiens, but others do not. Mizar toggles the global variables tracking this here. There is a common set of things toggled which we have isolated as the WEB macro <code>initialize_definition_item</code> common to initializing all definition items.

The correctness conditions are determined at this point, as well.

```
define initialize\_definition\_item \equiv gCorrectnessConditions \leftarrow []; gDefPos \leftarrow CurPos; gDefKind \leftarrow nItemKind 
 <math>\langle Global variables introduced in parseraddition.pas 1202 \rangle + \equiv gCorrectnessConditions: CorrectnessConditionsSet; 

1241. \langle Local variables for parser additions 1209 \rangle + \equiv
```

434 CONSTRUCTOR Mizar Parser §1242

```
1242.
        \langle \text{Initialize extended item by } ItemKind | 1230 \rangle + \equiv
itLociDeclaration:;
itDefMode: begin nItAllowed \leftarrow true; qExpandable \leftarrow false; initialize\_definition\_item end;
itDefAttr: begin initialize_definition_item end;
itAttrSynonym: begin initialize_definition_item end;
itAttrAntonym: begin initialize_definition_item end;
itModeNotation: begin initialize_definition_item end;
itDefFunc: begin nItAllowed \leftarrow true; initialize\_definition\_item end;
itFuncNotation: begin initialize_definition_item; end;
itDefPred, itPredSynonym, itCluster, itIdentify, itReduction:
  begin initialize_definition_item; end;
itPropertyRegistration: begin initialize\_definition\_item; qPropertySort \leftarrow PropertyKind(CurWord.Nr);
itDefStruct: begin initialize\_definition\_item; gStructPrefixes \leftarrow new(PList, Init(0)); end;
itCanceled: begin ErrImm(88); end;
1243. Correctness conditions. Registrations and definitions need correctness conditions to ensure the
well-definedness of adjective clusters and terms. The correctness conditions needed for a definition (or
registration) are inserted into the qCorrectnessConditions variable. When the correctness condition is found,
we remove it from the qCorrectnessConditions set.
\langle Global variables introduced in parseraddition.pas 1202 \rangle + \equiv
qRedefinitions: boolean;
1244. \langle Local variables for parser additions 1209 \rangle + \equiv
gCorrCondSort: CorrectnessKind;
\langle \text{Initialize extended item by } ItemKind | 1230 \rangle = itCorrCond:
         if CorrectnessKind(CurWord.Nr) \in qCorrectnessConditions then
    begin exclude(qCorrectnessConditions, CorrectnessKind(CurWord.Nr));
    gCorrCondSort \leftarrow CorrectnessKind(CurWord.Nr);
    if (qRedefinitions \land (qCorrCondSort = syCoherence) \land ExtBlockPtr(qBlockPtr) \uparrow .nHasAssumptions)
            then ErrImm(243);
    end
  else begin ErrImm(72); qCorrCondSort \leftarrow CorrectnessKind(0); end;
itCorrectness: if (qRedefinitions \land ExtBlockPtr(qBlockPtr) \uparrow .nHasAssumptions) then ErrImm(243);
1245.
        The last statement needing attention will be the scheme block. Note that gLocalScheme is not used
anywhere.
\langle \text{Local variables for parser additions } 1209 \rangle + \equiv
gLocalScheme: boolean;
gSchemePos: Position;
1246. (Initialize extended item by ItemKind 1230) +\equiv
itDefinition, itSchemeHead, itReservation, itPrivFuncDefinition, itPrivPredDefinition, itConstantDefinition,
       itExemplification: ;
itCaseBlock:;
itSchemeBlock: begin gLocalScheme \leftarrow CurWord.Kind \neq sy\_Scheme; gSchemePos \leftarrow CurPos; end;
1247. Popping an extended item.
\langle Global variables introduced in parseraddition.pas 1202 \rangle + \equiv
qSchemeParams: PList;
```

§1248 Mizar Parser CONSTRUCTOR 435

```
 \begin{array}{ll} \textbf{1248.} & \langle \operatorname{Local} \operatorname{variables} \ \operatorname{for} \ \operatorname{parser} \ \operatorname{additions} \ 1209 \, \rangle + \equiv \\ gPatternPos: \ Position; \\ gPattern: \ PatternPtr; \\ gNewPatternPos: \ Position; \\ gNewPattern: \ PatternPtr; \\ gSchemeIdNr: \ integer; \\ gSchemeIdPos: \ Position; \\ gSchemeConclusion: \ FormulaPtr; \\ gSchemePremises: \ PList; \\ \end{array}
```

Subsection 23.2.2. Popping

1249. Popping an item is invoked as part of *KillItem*, which occurs whenever (1) a semicolon is encountered, or (2) when starting a proof environment.

The contract for popping an item ensures the nContent field shall be populated for valid items.

NOTE: PASCAL has a set operation include(set, element) which adjoins an element to a set.

```
⟨Extended item implementation 1226⟩ +≡
procedure extItemObj.Pop;
var k: integer;
begin gLastWSItem↑.nItemEndPos ← PrevPos; ⟨Check for errors with definition items 1253⟩
⟨Update content of nLastWSItem based on type of item popped 1250⟩;
⟨Check the popped item's linkages are valid 1276⟩;
if gDefiningWay ≠ dfEmpty then
begin if gDefiniens↑.nDefSort = ConditionalDefiniens then
include(gCorrectnessConditions, syConsistency);
if gRedefinitions then include(gCorrectnessConditions, syCompatibility);
end;
inherited Pop; {(§704)}
end;
```

436 POPPING Mizar Parser $\S1250$

```
We will update the caller's nLastWSItem's contents in most cases.
\langle \text{Update content of } nLastWSItem \text{ based on type of item popped } 1250 \rangle \equiv
    case nItemKind of
    itTheorem: nLastWSItem \uparrow. nContent \leftarrow new(CompactStatementPtr, Init(new(PropositionPtr,
                  Init(nLabel, gLastFormula, nPropPos)), nInference));
    (Pop a proof step 1254)
    itConclusion, itRegularStatement: (Pop a conclusion or regular statement 1261)
    itGeneralization, itLociDeclaration: (Pop a "let" statement 1263)
    ⟨ Pop a definition item 1264⟩
    itPredSynonym, itPredAntonym, itFuncNotation, itModeNotation, itAttrSynonym, itAttrAntonym:
                  nLastWSItem \uparrow. nContent \leftarrow new(NotationDeclarationPtr, Init(gNewPatternPos, nItemKind,
                  gNewPattern, gPattern));
    ⟨ Pop a registration item 1272⟩
    itCorrCond: nLastWSItem \uparrow. nContent \leftarrow new(CorrectnessConditionPtr, Init(nItemPos, qCorrCondSort,
                  nInference));
    itCorrectness: nLastWSItem \uparrow .nContent \leftarrow new(CorrectnessConditionsPtr, Init(nItemPos,
                  gCorrectnessConditions, nInference));
    itProperty: nLastWSItem \uparrow. nContent \leftarrow new(PropertyPtr, Init(nItemPos, gPropertySort, nInference));
    itSchemeHead: nLastWSItem \uparrow .nContent \leftarrow new(SchemePtr, Init(qSchemeIdNr, qSchemeIdPos, qSchemeIdPo
                  gSchemeParams, gSchemePremises, gSchemeConclusion));
    ⟨ Pop skips remaining cases 1251⟩
    endcases
This code is used in section 1249.
1251. \langle \text{Pop skips remaining cases } 1251 \rangle \equiv
itPrivFuncDefinition, itPrivPredDefinition, itPragma, itDefinition, itSchemeBlock, itReservation,
             itExemplification, itCaseBlock:;
This code is used in section 1250.
                Check for errors. We need to flag a 253 or 254 error when the user tries to introduce an axiom
(which shouldn't occur much anymore, since axioms are not even documented anywhere).
\langle \text{Local variables for parser additions } 1209 \rangle + \equiv
gMeansPos: Position;
               \langle Check for errors with definition items 1253 \rangle \equiv
    case nItemKind of
    itDefPred, itDefFunc, itDefMode, itDefAttr: begin if qDefiningWay \neq dfEmpty then
             begin if nDefiniensProhibited \land \neg AxiomsAllowed then
                 begin Error(gMeansPos, 254); gDefiningWay \leftarrow dfEmpty; end;
             end
         else if \neg qRedefinitions \land \neg nDefiniensProhibited \land \neg AxiomsAllowed then SemErr(253);
         end;
    endcases;
This code is used in section 1249.
```

 $\S1254$ Mizar Parser POPPING 437

1254. Pop a proof step. Popping a proof step should assign to the contents of the caller's *nLastWsItem* some kind of inference justification, usually in the form of a statement in the WSM syntax tree.

```
\langle Pop a proof step 1254 \rangle \equiv itPerCases: nLastWSItem \uparrow .nContent \leftarrow nInference; See also sections 1255, 1257, and 1259.
This code is used in section 1250.
```

1255. Popping a reconsideration. We should assign a *TypeChangingStatement* to the content of the caller's last item, using the *nInference* field of the caller as the justification.

```
\langle \text{Pop a proof step } 1254 \rangle + \equiv itReconsider: nLastWSItem \uparrow .nContent \leftarrow new(TypeChangingStatementPtr, Init(gReconsiderList, gLastType, SimpleJustificationPtr(nInference)));
```

1256. Popping existential elimination and introduction. We assign a consider (or given) WSM statement to the caller's previous *WSItem*'s contents when popping a choice (resp., existential assumption) item.

We should remind the reader of the grammar here:

```
\langle \mathit{Qualified}\text{-}\mathit{Segment} \rangle ::= \langle \mathit{Variables} \rangle \langle \mathit{Qualification} \rangle
\langle \mathit{Variables} \rangle ::= \langle \mathit{Variable} \rangle \{ \text{","} \langle \mathit{Variable} \rangle \}
\langle \mathit{Qualification} \rangle ::= (\text{"being"} \mid \text{"be"}) \langle \mathit{Type}\text{-}\mathit{Expression} \rangle
```

And, of course, a qualified-segment list is just a comma-separated list of qualified-segments.

```
\langle Global variables introduced in parseraddition.pas 1202\rangle +\equiv gQualifiedSegmentList: PList;
```

```
1257. \langle \text{Pop a proof step } 1254 \rangle + \equiv
```

```
itChoice: \mathbf{begin} \ nLastWSItem \uparrow. nContent \leftarrow new(ChoiceStatementPtr, Init(gQualifiedSegmentList, gPremises, SimpleJustificationPtr(nInference))); gPremises \leftarrow \mathbf{nil};
```

```
itExistentialAssumption: \ \mathbf{begin} \ nLastWSItem \uparrow .nContent \leftarrow new (ExistentialAssumptionPtr, \\ Init (nItemPos, gQualifiedSegmentList, gPremises)); \ gPremises \leftarrow \mathbf{nil}; \\ \mathbf{end};
```

1258. Popping a stipulation. When we pop a case, suppose, or assume — some kind of "assumption"-like statement — we are assigning either a *CollectiveAssumption* object or a *SingleAssumption* object to the content of the *current WSItem* global variable.

```
\langle Local variables for parser additions 1209\rangle +\equiv gThatPos: Position;
```

```
1259. \langle \text{Pop a proof step } 1254 \rangle + \equiv
```

```
itSupposeHead, itCaseHead, itAssumption: if gPremises \neq nil then begin gLastWSItem\uparrow.nContent \leftarrow new(CollectiveAssumptionPtr, Init(gThatPos, gPremises)); gPremises \leftarrow nil; end
```

```
\textbf{else} \ \ gLastWSItem \uparrow. nContent \leftarrow new(SingleAssumptionPtr, Init(nItemPos, new(PropositionPtr, Init(nLabel, gLastFormula, nPropPos)))); \\
```

438 POPPING Mizar Parser $\S1260$

1260. Pop a conclusion or regular statement. We assign an appropriate WSM statement node to the previous item's contents.

1262. Pop a 'let' statement. For generic let statements of the form

This code is used in section 1250.

```
let \vec{x}_1 be T_1, \ldots, \vec{x}_n be T_n
```

we transform it to n statements of the form "let \vec{x} be T", then add these to the gLastWSBlock's items. When we have

let \vec{x} be T such that Φ

```
we need to add a Collective Assumption node to the global gLast WSBlock's items.
\langle Local variables for parser additions 1209\rangle + \equiv
gSuchPos: Position;
1263. \langle \text{Pop a "let" statement 1263} \rangle \equiv
  begin for k \leftarrow 0 to gQualifiedSegmentList \uparrow. Count - 1 do
     begin qLastWSItem \leftarrow qWsTextProper \uparrow.NewItem(nItemKind,
           QualifiedSegmentPtr(gQualifiedSegmentList \uparrow . Items \uparrow [k]) \uparrow . nSegmPos);
     nLastWSItem \leftarrow qLastWSItem; qLastWSItem \uparrow .nContent \leftarrow qQualifiedSegmentList \uparrow .Items \uparrow [k];
     if k = gQualifiedSegmentList \uparrow. Count - 1 then gLastWSItem \uparrow. nItemEndPos \leftarrow PrevPos
     else gLastWSItem\uparrow.nItemEndPos \leftarrow QualifiedSegmentPtr(gQualifiedSegmentList\uparrow.Items\uparrow[k+
              1])\uparrow.nSegmPos;
     qQualifiedSegmentList \uparrow . Items \uparrow [k] \leftarrow nil; qLastWSBlock \uparrow . nItems . Insert (qLastWSItem);
     end:
  dispose(gQualifiedSegmentList, Done);
  if gPremises \neq nil then
     begin gLastWSItem \leftarrow gWsTextProper \uparrow.NewItem(itAssumption, gSuchPos);
     gLastWSItem \uparrow .nContent \leftarrow new(CollectiveAssumptionPtr, Init(gThatPos, gPremises));
     qPremises \leftarrow nil; qLastWSItem\uparrow.nItemEndPos \leftarrow PrevPos; nLastWSItem \leftarrow qLastWSItem;
     qLastWSBlock \uparrow .nItems.Insert(qLastWSItem);
     end;
  end;
```

 $\{1264$ Mizar Parser POPPING 439

1264. Pop a mode definition. A mode is either expandable (an abbreviation) or nonexpandable. For expandable modes, we just add a new *ExpandableModeDefinition* WSM object to the caller's *nLastWSItem*'s contents.

On the other hand, non-expandable modes should add to the caller's nLastWSItem's contents a new StandardModeDefinition object. If this is not a redefinition, then we must add the "existence" correctness condition to the global variable qCorrectnessConditions.

```
condition to the global variable gCorrectnessConditions.
\langle \text{ Pop a definition item } 1264 \rangle \equiv
itDefMode: begin if gExpandable then nLastWSItem \uparrow .nContent \leftarrow new(ExpandableModeDefinitionPtr,
          Init(gPatternPos, ModePatternPtr(gPattern), gLastType))
  else begin nLastWSItem \uparrow .nContent \leftarrow new(StandardModeDefinitionPtr, Init(qPatternPos,
          gRedefinitions, ModePatternPtr(gPattern), gSpecification, gDefiniens));
     if \neg gRedefinitions then include(gCorrectnessConditions, syExistence);
     end:
  end;
See also sections 1265, 1266, 1267, and 1270.
This code is used in section 1250.
1265. Pop a functor definition. When popping a functor definition, we just add a FunctorDefinition
object to the caller's nLastWSItem's contents.
\langle \text{ Pop a definition item } 1264 \rangle + \equiv
itDefFunc: \mathbf{begin} \ nLastWSItem \uparrow .nContent \leftarrow new(FunctorDefinitionPtr, Init(qPatternPos,
       qRedefinitions, FunctorPatternPtr(qPattern), qSpecification, qDefiningWay, qDefiniens));
  end;
1266. Pop an attribute definition. We just need to add an AttributeDefinition object to the caller's
nLastWSItem's contents.
\langle \text{ Pop a definition item } 1264 \rangle + \equiv
itDefAttr: \mathbf{begin} \ nLastWSItem \uparrow .nContent \leftarrow new(AttributeDefinitionPtr, Init(gPatternPos,
       gRedefinitions, AttributePatternPtr(gPattern), gDefiniens));
  end;
1267. Pop a predicate definition. We just need to add a PredicateDefinition object to the caller's
nLastWSItem's contents.
\langle \text{ Pop a definition item } 1264 \rangle + \equiv
itDefPred: \mathbf{begin} \ nLastWSItem \uparrow. nContent \leftarrow new(PredicateDefinitionPtr, Init(qPatternPos,
       gRedefinitions, PredicatePatternPtr(gPattern), gDefiniens));
  end;
1268. Popping a structure definition. We just need to add a Structure Definition object to the caller's
nLastWSItem's contents.
\langle Global variables introduced in parseraddition.pas 1202 \rangle + \equiv
qConstructorNr: integer;
1269. \langle Local variables for parser additions 1209 \rangle + \equiv
qParams: PList;
gStructFields: PList;
1270. \langle \text{Pop a definition item } 1264 \rangle + \equiv
```

itDefStruct: **begin** $nLastWSItem \uparrow .nContent \leftarrow new(StructureDefinitionPtr, Init(qPatternPos,$

gStructPrefixes, gConstructorNr, gParams, gStructFields));

end;

440 POPPING Mizar Parser $\S1271$

Pop a cluster registration item. A "cluster" registration (i.e., a existential, conditional, or functor registration) adds to the caller's nLastWSItem's contents a new cluster object (of appropriate kind). The qClusterSort is populated when the parser finishes a cluster registration when invoking extItemObj. FinishAntecedent ■ $(\S1281)$ or similar methods. The gClusterTerm is populated in the extItemObj.FinishClusterTerm method (§1282). $\langle \text{Local variables for parser additions } 1209 \rangle + \equiv$ gAntecedent, gConsequent: PList; gClusterTerm: TermPtr;1272. $\langle \text{Pop a registration item } 1272 \rangle \equiv$ itCluster: begin case gClusterSort of ExistentialRegistration: begin $nLastWSItem \uparrow. nContent \leftarrow new(EClusterPtr, Init(nItemPos, gConsequent, gLastType));$ include(gCorrectnessConditions, syExistence)Conditional Registration: begin $nLastWSItem \uparrow .nContent \leftarrow new(CClusterPtr, Init(nItemPos,$ gAntecedent, gConsequent, gLastType); include(gCorrectnessConditions, syCoherence); end: FunctorialRegistration: begin $nLastWSItem \uparrow .nContent \leftarrow new(FClusterPtr, Init(nItemPos,$ gClusterTerm, gConsequent, gLastType); include(gCorrectnessConditions, syCoherence); end; endcases; end; See also section 1275. This code is used in section 1250. 1273. Pop a registration item. For an identify or reduce registration, we assign the content of the caller's nLastWSItem a new IdentifyRegistration (resp., ReduceRegistration) object. Identify registrations use the gIdentifyEqLociList local variable, while the reduction registrations use the gLeftTermInReduction module-wide variable. \langle Global variables introduced in parseraddition.pas $1202 \rangle + \equiv$ gLeftTermInReduction: TermPtr;1274. $\langle \text{Local variables for parser additions } 1209 \rangle + \equiv$ gIdentifyEqLociList: PList;**1275.** $\langle \text{Pop a registration item } 1272 \rangle + \equiv$ $itIdentify: \textbf{begin} \ nLastWSItem \uparrow. nContent \leftarrow new (IdentifyRegistrationPtr, Init(nItemPos, gNewPattern, and all the properties of the$ qPattern, qIdentifyEqLociList); include(qCorrectnessConditions, syCompatibility); end: itReduction: begin $nLastWSItem \uparrow .nContent \leftarrow new(ReduceRegistrationPtr, Init(nItemPos,$ qLeftTermInReduction, qLastTerm); include(qCorrectnessConditions, syReducibility); end:

 $itPropertyRegistration: SethoodRegistrationPtr(nLastWSItem \uparrow. nContent) \uparrow. nJustification \leftarrow nInference;$

§1276 **POPPING** 441 Mizar Parser

Check linkages are valid. When popping an item, we should check if the block containing the caller is nLinked. If so, flag a "178" error and assign $nLinked \leftarrow false$. Update the block's nLinkAlloweddepending on the caller's nLinkable field. But if the parser is in panic mode, the containing block's nLinkAllowed and nLinkProhibited are both assigned to false. [This configuration appears to encode a particular state which feels a bit of a "kludge" to me...

```
\langle Check the popped item's linkages are valid 1276 \rangle \equiv
  with extBlockPtr(gBlockPtr)\uparrow do
     begin if nLinked then
        begin Error(nLinkPos, 178); nLinked \leftarrow false end;
     nLinkAllowed \leftarrow nLinkable; nLinkProhibited \leftarrow \neg nLinkable;
     if \neg StillCorrect then
        begin nLinkAllowed \leftarrow false; nLinkProhibited \leftarrow false end;
     end
This code is used in section 1249.
```

Subsection 23.2.3. Registrations and notations

1277. Processing synonyms. We need to update the qNewPatternPos and qNewPattern global variables when processing a synonym.

```
define process\_notation\_item \equiv qNewPatternPos \leftarrow qPatternPos; qNewPattern \leftarrow qPattern
\langle Extended item implementation 1226\rangle + \equiv
procedure extItemObj.ProcessModeSynonym;
  begin process_notation_item; end;
procedure extItemObj.ProcessAttrSynonym;
  begin process_notation_item; end;
procedure extItemObj.ProcessAttrAntonym;
  begin process_notation_item; end;
procedure extItemObj.ProcessPredSynonym;
  begin process_notation_item; end;
procedure extItemObj.ProcessPredAntonym;
  begin process_notation_item; end;
procedure extItemObj.ProcessFuncSynonym;
  begin process_notation_item; end;
        Starting attributes. This is used when the parser encounters a cluster registration (§1643). The
gAttrColl is populated in the extSubexpObj.CompleteAdjectiveCluster (§1398) method.
\langle \text{Local variables for parser additions } 1209 \rangle + \equiv
gAttrColl: PList;
1279. \langle Extended item implementation 1226 \rangle + \equiv
procedure extItemObj.StartAttributes;
  begin gAttrColl \leftarrow new(PList, Init(6));
  end:
```

end;

1280. Starting a sentence. We just need to populate the caller's nPropPos, assigning to it the current position of the parser.

```
⟨ Extended item implementation 1226⟩ +≡ procedure extItemObj.StartSentence; begin nPropPos ← CurPos; end;
```

1281. Processing conditional registration. This populates the *gClusterSort* and the related global variables, as the parser finishes parsing the antecedent and consequent to the cluster.

```
⟨ Extended item implementation 1226⟩ +≡
procedure extItemObj.FinishAntecedent;
begin gClusterSort ← ConditionalRegistration; gAntecedent ← gAttrColl;
end;
procedure extItemObj.FinishConsequent;
begin gConsequent ← gAttrColl;
end;

1282. Finishing a cluster. This populates the gClusterSort and the gClusterTerm.
⟨ Extended item implementation 1226⟩ +≡
procedure extItemObj.FinishClusterTerm;
```

begin $qClusterSort \leftarrow FunctorialRegistration; <math>qClusterTerm \leftarrow qLastTerm;$

1283. Identify registration. Schematically, we have the registration statement look like (using global variable names for the subexpressions):

```
identify \langle gNewPattern \rangle with \langle gPattern \rangle [when \langle gIdentifyEqLociList \rangle];
```

We store the first pattern in the gNewPattern global variable, then the second pattern in the gPattern global variable. Completing the identify registration will check if the current word is "when" and, if so, start a list of loci equalities.

```
\langle Extended item implementation 1226\rangle += procedure extItemObj.StartFuncIdentify; begin end; procedure extItemObj.ProcessFuncIdentify; begin gNewPatternPos \leftarrow gPatternPos; gNewPattern \leftarrow gPattern; end; procedure extItemObj.CompleteFuncIdentify; begin gIdentifyEqLociList \leftarrow nil; if CurWord.Kind = sy\_When then gIdentifyEqLociList \leftarrow new(PList,Init(0)); end;
```

1284. "Reduces to" registrations. Recall, these schematically look like

```
reduce \langle gLeftLocus \rangle to \langle Locus \rangle;
```

Mizar will populate gLeftLocus. The gambit will be to treat this as a functor pattern; i.e., the gLeftLocus will be used to populate gNewPattern in the method extItemObj.FinishFunctorPattern (§1323).

```
\langle Local variables for parser additions 1209\rangle +\equiv gLeftLocus: LocusPtr;
```

```
1285. \langle \text{Extended item implementation } 1226 \rangle + \equiv

procedure extItemObj.ProcessLeftLocus;

begin gLeftLocus \leftarrow new(LocusPtr, Init(CurPos, GetIdentifier));

end;

procedure extItemObj.ProcessRightLocus;

begin gIdentifyEqLociList.Insert(new(LociEqualityPtr, Init(PrevPos, gLeftLocus, new(LocusPtr, Init(CurPos, GetIdentifier)))));

end;

procedure extItemObj.StartFuncReduction;

begin end;

procedure extItemObj.ProcessFuncReduction;

begin gNewPatternPos \leftarrow gPatternPos; gLeftTermInReduction \leftarrow gLastTerm;

end;
```

Subsection 23.2.4. Processing definitions

1286. The terminology used by the parser appears to be ($\S\S1565 \ et \ seq.$):

```
let ⟨Fixed Variables⟩;
```

and

```
consider \langle Fixed\ Variables \rangle such that...
```

This would mean that we would have "fixed variables" refer to a list of qualified segments. We remind the reader of the grammar

```
\langle Fixed\text{-}Variables \rangle ::= \langle Implicitly\text{-}Qualified\text{-}Variables \rangle \ \{ \text{ "," } \langle Fixed\text{-}Variables \rangle \} \\ | \langle Explicitly\text{-}Qualified\text{-}Variables \rangle \ \{ \text{ "," } \langle Fixed\text{-}Variables \rangle \} \\ \langle Implicitly\text{-}Qualified\text{-}Variables \rangle ::= \langle Variables \rangle \\ \langle Explicitly\text{-}Qualified\text{-}Variables \rangle ::= \langle Qualified\text{-}Segment \rangle \ \{ \text{ "," } \langle Qualified\text{-}Segment \rangle \} \\ \langle Qualified\text{-}Segment \rangle ::= \langle Variables \rangle \ \langle Qualification \rangle \\ \langle Variables \rangle ::= \langle Variable \rangle \ \{ \text{ "," } \langle Variable \rangle \} \\ \langle Qualification \rangle ::= (\text{"be" } | \text{"being"}) \ \langle Type \rangle
```

The "fixed variables" routine in the parser will parse a comma-separated list of qualified variables.

CAUTION: The grammar in the syntax.txt file is actually more strict than this, because it actually states the following:

```
\langle Loci\text{-}Declaration \rangle ::= "let" \langle Qualified\text{-}Variables \rangle [ "such" \langle Conditions \rangle ] ;
```

The grammar for a qualified segment requires implicitly qualified variables appear at the very end.

```
\langle Extended item implementation 1226\rangle += procedure extItemObj.StartFixedVariables; begin gQualifiedSegmentList \leftarrow new(PList, Init(0)); end;
```

```
1287. \langle Global variables introduced in parseraddition.pas 1202 \rangle += gQualifiedSegment: MList; gSegmentPos: Position;
```

1288. Fixed segments. This refers to each "explicitly qualified segment" or "implicitly qualified segment" appearing in the fixed variables portion. The fixed segments are separated by commas.

```
\langle Extended item implementation 1226\rangle += procedure extItemObj.StartFixedSegment; begin gQualifiedSegment.Init(0); gSegmentPos \leftarrow CurPos; end;
```

1289. When parsing fixed variables, and the parser has just entered the loop to parse fixed variables, this function will be invoked.

```
⟨ Extended item implementation 1226⟩ +≡
procedure extItemObj.ProcessFixedVariable;
begin gQualifiedSegment.Insert(new(VariablePtr, Init(CurPos, GetIdentifier)));
end;
```

1290. This "clears the cache" for assigning the type in an explicitly qualified segment (appearing in a fixed variable segment).

```
\langle Extended item implementation 1226\rangle +\equiv procedure extItemObj.ProcessBeing; begin gLastType \leftarrow nil; end;
```

1291. The last statement in the parser loop when parsing "fixed variables" is to push the "fixed segment" onto the gQualifiedSegmentList global variable. There are two cases to consider: the implicitly qualified variables and the explicitly qualified variables.

The implicitly qualified case simple moves the pointers around "manually", so we need to update every entry of gQualifiedSegment.Items to be nil. The explicitly qualified case moves the pointers around using the MList constructor, mutating gQualifiedSegment into a list of nil pointers.

1292. When we finish parsing fixed variables, we need to "unset" the *gPremises* global variable. The parser will either be looking at a semicolon token or at "such $\langle Conditions \rangle$ ". The reader should note that gSuchThatOcc is not used in the parser, nor anywhere else in Mizar. But we recall (§1262) the gSuchPos is used when popping a let statement.

```
\langle Local \text{ variables for parser additions } 1209 \rangle + \equiv gSuchThatOcc: boolean; { not used }
```

```
1293. \langle \text{Extended item implementation } 1226 \rangle + \equiv procedure extItemObj.FinishFixedVariables; begin gSuchThatOcc \leftarrow CurWord.Kind = sy\_Such; gSuchPos \leftarrow CurPos; gPremises \leftarrow \mathbf{nil}; end;
```

1294. When the parser encounters the statement:

```
let \langle Fixed\text{-}Variables \rangle such that \langle Assumption \rangle;
```

The first things it does when encountering the "such" token is move to the next token ("that") and then invoke the *StartAssumption* method. We should allocate a fresh list for *gPremises* and mark the position of the "that" token.

```
\langle \text{ Extended item implementation } 1226 \rangle + \equiv
procedure extItemObj.StartAssumption;
begin gPremises \leftarrow new(PList,Init(0)); gThatPos \leftarrow CurPos;
end:
```

1295. Finishing an assumption will update the global variable gBlockPtr's field reflecting it has assumptions.

```
\langle Extended item implementation 1226\rangle += procedure extItemObj.FinishAssumption; begin ExtBlockPtr(gBlockPtr)\uparrow.nHasAssumptions \leftarrow true; end;
```

1296. When the Mizar parser has encountered

```
assume that \langle Conditions \rangle;
```

we start a collective assumption when the parser has just encountered the "that" token. As with the "let statement with assumptions", we need to allocate a new list for gPremises and assign the gThatPos to the current position.

```
\langle \text{ Extended item implementation } 1226 \rangle + \equiv
procedure extItemObj.StartCollectiveAssumption;
begin gPremises \leftarrow new(PList,Init(0)); gThatPos \leftarrow CurPos;
end;
```

1297. Processing copula in a definition. When defining a (nonexpandable) mode, a functor, a predicate, or an attribute, we have

```
\langle Pattern \rangle means \langle Expression \rangle;
```

or

```
\langle Pattern \rangle equals \langle Expression \rangle;
```

The expression may or may not be labeled, we may or may not have the definition-by-cases. Whatever the situation, we should initialize the variables describing the definiens:

- the *gDefLabId* should be reset to zero (and populated in the *ProcessDefLabel* method);
- the gDefLabPos should be reset to the current position (and populated in the ProcessDefLabel method);
- \bullet the *gDefiningWay* should be assigned to *dfMeans* or *dfEquals* depending on the copula used in the definition;
- the *gOtherwise* pointer should be assigned to **nil**;
- the gMeansPos position should be assigned to the current position.

Following tradition in logic, we will refer to "means" and "equals" as the "Copula" in the definition.

```
\langle\, \text{Global variables introduced in parseraddition.pas 1202}\,\rangle\,+\!\equiv\,gDefLabId\colon\,integer;\,gDefLabPos\colon\,Position;
```

1298. $\langle \text{Local variables for parser additions 1209} \rangle + \equiv gOtherwise: PObject;$

```
1299. \langle \text{Extended item implementation } 1226 \rangle + \equiv procedure extItemObj.ProcessMeans; begin gDefLabId \leftarrow 0; gDefLabPos \leftarrow CurPos; gDefiningWay \leftarrow dfMeans; gOtherwise \leftarrow nil; gMeansPos \leftarrow CurPos
```

procedure *extItemObj.ProcessEquals*;

end;

```
begin gDefLabId \leftarrow 0; gDefLabPos \leftarrow CurPos; gDefiningWay \leftarrow dfEquals; gOtherwise \leftarrow \mathbf{nil}; gMeansPos \leftarrow CurPos; end:
```

1300. When parsing a definition-by-cases, the cases are terminated with an "otherwise" keyword. Recall the grammar for such definitions looks like:

```
\langle Partial\text{-}Definiens\text{-}List \rangle "otherwise" \langle Expression \rangle;
```

What happens depends on whether the definition uses "means" or "equals": in the former case, we should update the gOtherwise pointer to be the gLastFormula; in the latter case, we should update the gOtherwise to be the gLastTerm.

```
\langle Extended item implementation 1226\rangle +\equiv procedure extItemObj.FinishOtherwise; begin if gDefiningWay = dfEquals then gOtherwise \leftarrow gLastTerm else gOtherwise \leftarrow gLastFormula; end;
```

§1301 Mizar Parser PROCESSING DEFINITIONS 447

1301. Starting a definiens should mutate the $it_Allowed$ global variable to be equal to the caller's nItAllowed field. The $it_Allowed$ global variable is toggled on and off when the parser encounters "guards" in conditional definitions, whereas the nItAllowed fields reflects whether the sort of definition allows "it" in the definiens.

```
⟨ Extended item implementation 1226⟩ +≡ procedure extItemObj.StartDefiniens; begin it_Allowed ← nItAllowed; end;
```

1302. "Guards" refers to the conditions in a definition-by-cases. Specifically, we have

```
\langle Partial\text{-}Definiens \rangle ::= \langle Expression \rangle "if" \langle Guard\text{-}Formula \rangle
```

be the grammar for one particular case. We have a comma-separated list of partial definiens, so whenever the parser (a) first encounters the "if" keyword in a definiens, or (b) has already encountered the "if" keyword and now has encountered a comma — these are the two cases to start a new guard.

```
\langle Local variables for parser additions 1209\rangle += gPartDef: PObject;
```

```
1303. \langle \text{Extended item implementation } 1226 \rangle +\equiv  procedure extItemObj.StartGuard; begin if gPartialDefs =  nil then gPartialDefs \leftarrow new(PList, Init(0)); it\_Allowed \leftarrow false; if gDefiningWay = dfMeans then gPartDef \leftarrow gLastFormula else gPartDef \leftarrow gLastTerm; end:
```

1304. After parsing a formula, then the parser will invoke FinishGuard. This will append to gPartialDefs a new partial definiens.

```
\langle \text{ Extended item implementation } 1226 \rangle + \equiv \\ \textbf{procedure } extItemObj.FinishGuard; \\ \textbf{begin } it\_Allowed \leftarrow nItAllowed; \\ \textbf{case } gDefiningWay \textbf{ of} \\ dfMeans: gPartialDefs.Insert(new(PartDefPtr,Init(new(DefExpressionPtr,Init(exFormula, gPartDef)), gLastFormula)));} \\ dfEquals: gPartialDefs.Insert(new(PartDefPtr,Init(new(DefExpressionPtr,Init(exTerm,gPartDef)), gLastFormula)));} \\ \textbf{endcases;} \\ \textbf{end:} \\ \\ \end{aligned}
```

1305. Recall for functor definitions we have something like:

```
func \langle Pattern \rangle \rightarrow \langle Type \rangle (means | equals ) ...
```

Similarly, nonexpandable modes look like

$$\verb|mode| \langle Pattern \rangle -> \langle Type \rangle \> \verb|means| \ldots$$

The "-> $\langle Type \rangle$ " is called the [type] specification for the definition. We should update the gSpecification global variable to point to whatever the last type parsed was — which is stored in the gLastType global variable.

```
\langle Extended item implementation 1226\rangle +\equiv procedure extItemObj.FinishSpecification; begin gSpecification \leftarrow gLastType; end;
```

1306. "Construction type" is the term used by the parser for "nonexpandable modes". They, too, have a type specification. The FinishConstructionType populates the gSpecification global variable with this type.

```
\langle Extended item implementation 1226\rangle +\equiv procedure extItemObj.FinishConstructionType;
```

```
begin gSpecification \leftarrow gLastType; end;
```

1307. Expandable mode definitions, after encountering the "is" keyword, invokes the *StartExpansion* method. This just ensures there is no definiens, and the *gExpandable* global variable is assigned to "true".

```
\langle Extended item implementation 1226\rangle +\equiv procedure extItemObj.StartExpansion; begin if gRedefinitions then ErrImm(271); nDefiniensProhibited \leftarrow true; gExpandable \leftarrow true; end;
```

1308. The parser, when determining the pattern for an attribute (§1608), resets the state when starting to determine the pattern for the attribute. This is handled by the *StartAttribute* method.

We should remind the reader that attributes can only have arguments to its left.

```
\langle \, \text{Global variables introduced in parseraddition.pas 1202} \, \rangle \, + \equiv \, gParamNbr: \, integer;
```

```
1309. \langle \text{Local variables for parser additions 1209} \rangle + \equiv gLocus: LocusPtr;
```

```
1310. ⟨Extended item implementation 1226⟩ +≡ procedure extItemObj.StartAttributePattern;
```

```
begin gParamNbr \leftarrow 0; gParams \leftarrow nil; gLocus \leftarrow new(LocusPtr, Init(CurPos, GetIdentifier)); end;
```

1311. Since an attribute can only have attributes to its left, it's pretty clear when the attribute pattern has been parsed: the parser has found the attribute being defined. In that case (assuming we're not panicking), we should add the attribute format to the *qFormatsColl* dictionary and update the global variables.

```
\langle \text{ Extended item implementation } 1226 \rangle +\equiv \\ \textbf{procedure } extItemObj.FinishAttributePattern; \\ \textbf{var } lFormatNr: integer; \\ \textbf{begin } lFormatNr \leftarrow 0; \\ \textbf{if } (CurWord.Kind = AttributeSymbol) \land stillcorrect \textbf{ then} \\ lFormatNr \leftarrow gFormatsColl.CollectPrefixForm(`V`, CurWord.Nr, gParamNbr); \\ gPatternPos \leftarrow CurPos; gConstructorNr \leftarrow CurWord.Nr; \\ gPattern \leftarrow new(AttributePatternPtr, Init(gPatternPos, gLocus, gConstructorNr, gParams)); \\ \textbf{end}; \\ \end{aligned}
```

1312. A mode definition may include a "sethood" property. This particular function is used when registering sethood in a registration block.

```
\langle Extended item implementation 1226\rangle += procedure extItemObj.FinishSethoodProperties; begin nLastWSItem\uparrow.nContent \leftarrow new(SethoodRegistrationPtr, Init(nItemPos, gPropertySort, gLastType)); end;
```

1313. We remind the reader the grammar for a mode pattern

```
\langle Mode\text{-}Pattern \rangle ::= \langle Mode\text{-}Symbol \rangle \ [ "of" \langle Loci \rangle \ ]
```

The loci parameters can only appear *after* the mode symbol (and before the "of" reserved keyword). Starting a mode pattern should reset the relevant global variables.

```
\langle Extended item implementation 1226\rangle +\equiv procedure extItemObj.StartModePattern; begin gParamNbr \leftarrow 0; gParams \leftarrow \mathbf{nil}; gPatternPos \leftarrow CurPos; gConstructorNr \leftarrow CurWord.Nr; end;
```

1314. Finishing a mode pattern should build a new ModePatternObj, and store it in the gPattern global variable. And if we are not panicking, we should add it to the gFormatsColl dictionary.

```
\langle \text{ Extended item implementation } 1226 \rangle + \equiv \\ \textbf{procedure } extItemObj.FinishModePattern; \\ \textbf{var } lFormatNr: integer; \\ \textbf{begin } lFormatNr \leftarrow 0; \\ \textbf{if } StillCorrect \textbf{ then } lFormatNr \leftarrow gFormatsColl.CollectPrefixForm(`M`, gConstructorNr, gParamNbr); \\ gPattern \leftarrow new(ModePatternPtr, Init(gPatternPos, gConstructorNr, gParams)); \\ \textbf{end}; \\ \end{cases}
```

1315. When parser starts parsing a new predicate pattern, we should reset the relevant global variables.

```
\langle Extended item implementation 1226\rangle +\equiv procedure extItemObj.StartPredicatePattern; begin gParamNbr \leftarrow 0; gParams \leftarrow nil; end;
```

1316. When the parser tries to parse a "predicative formula" (i.e., a formula involving a predicate) — including predicate patterns — the first thing it does is invoke this *ProcessPredicateSymbol* method. This resets the global variables needed to populate the arguments to the predicate in the formula.

```
\langle Global variables introduced in parseraddition.pas 1202 \rangle += gLeftLociNbr: integer;
```

1317. $\langle \text{Local variables for parser additions 1209} \rangle + \equiv gLeftLoci: PList;$

```
1318. ⟨Extended item implementation 1226⟩ +≡ procedure extItemObj.ProcessPredicateSymbol;
```

```
 \begin{array}{l} \mathbf{begin} \ gPatternPos \leftarrow CurPos; \ gLeftLociNbr \leftarrow gParamNbr; \ gLeftLoci \leftarrow gParams; \ gParamNbr \leftarrow 0; \\ gParams \leftarrow \mathbf{nil}; \ gConstructorNr \leftarrow CurWord.Nr; \\ \mathbf{end}; \end{array}
```

1319. Finishing a predicate pattern will create a new PredicatePattern object, update the gPattern global variable to point to it, and (if the parser is not panicking) add the predicate's format to the gFormatsColl dictionary.

```
\langle \text{ Extended item implementation } 1226 \rangle +\equiv \\ \textbf{procedure } extItemObj.FinishPredicatePattern; \\ \textbf{var } lFormatNr: integer; \\ \textbf{begin } lFormatNr \leftarrow 0; \\ \textbf{if } StillCorrect \textbf{ then} \\ lFormatNr \leftarrow gFormatsColl.CollectPredForm(gConstructorNr, gLeftLociNbr, gParamNbr); \\ gPattern \leftarrow new(PredicatePatternPtr, Init(gPatternPos, gLeftLoci, gConstructorNr, gParams)); \\ \textbf{end;} \end{aligned}
```

1320. Functor patterns a bit trickier. When starting one, what should occur depends on the type of functor being defined. Specifically, we handle brackets differently than other functors, and within the brackets we handle braces (i.e., definitions like $\{x_1, \ldots, x_n\}$) differently than square brackets ($[x_1, \ldots, x_n]$) differently than everything other functor bracket.

In all cases, even non-bracket functors, we need to reset the gParamNbr and gParams global variables so they may be populated correctly.

```
\langle Global variables introduced in parseraddition.pas 1202\rangle +\equiv gSubItemKind: TokenKind;
```

```
1321. \langle Extended item implementation 1226\rangle + \equiv procedure extItemObj.StartFunctorPattern;
```

```
begin gPatternPos \leftarrow CurPos; gSubItemKind \leftarrow CurWord.Kind; case CurWord.Kind of 
 LeftCircumfixSymbol: gConstructorNr \leftarrow CurWord.Nr; sy\_LeftSquareBracket: begin gSubItemKind \leftarrow LeftCircumfixSymbol; gConstructorNr \leftarrow SquareBracket end; sy\_LeftCurlyBracket: begin gSubItemKind \leftarrow LeftCircumfixSymbol; gConstructorNr \leftarrow CurlyBracket end; othercases gConstructorNr \leftarrow 0; endcases; gParamNbr \leftarrow 0; gParams \leftarrow nil; end;
```

§1322 Mizar Parser PROCESSING DEFINITIONS 451

1322. For "non-bracket" functors (i.e., infix operators), the functor pattern is processed by (1) getting the left parameters, (2) processing the functor symbol, (3) getting the right parameters. This function is precisely step (2).

```
 \langle \text{ Extended item implementation } 1226 \rangle + \equiv \\ \textbf{procedure } extItemObj.ProcessFunctorSymbol; \\ \textbf{begin } gPatternPos \leftarrow CurPos; \\ \textbf{if } CurWord.Kind = InfixOperatorSymbol \ \textbf{then} \\ \textbf{begin } gSubItemKind \leftarrow InfixOperatorSymbol; \ gConstructorNr \leftarrow CurWord.Nr; \\ gLeftLociNbr \leftarrow gParamNbr; \ gLeftLoci \leftarrow gParams; \ gParamNbr \leftarrow 0; \ gParams \leftarrow \textbf{nil}; \\ \textbf{end}; \\ \textbf{end}; \\ \textbf{end};
```

1323. When defining a bracket functor pattern, we add a new bracket format to the gFormatsColl dictionary, and then set gPattern to a newly allocated Bracket pattern.

When defining an infix functor, we add a new functor format to the gFormatsColl dictionary, and then we set the gPattern to a newly allocated infix functor pattern.

The "other cases" constructs an infix functor pattern, but does not add the form to the gFormatsColl dictionary.

```
\langle \text{Extended item implementation } 1226 \rangle + \equiv
procedure extItemObj.FinishFunctorPattern;
  var lConstructorNr, lFormatNr: integer;
  begin lFormatNr \leftarrow 0;
  case gSubItemKind of
  LeftCircumfixSymbol: begin lConstructorNr \leftarrow CurWord.Nr;
    if StillCorrect then
       lFormatNr \leftarrow qFormatsColl.CollectBracketForm(qConstructorNr, lConstructorNr, qParamNbr, 0, 0);
    qPattern \leftarrow new(CircumfixFunctorPatternPtr, Init(qPatternPos, qConstructorNr, lConstructorNr, lConstructorNr)
         gParams));
    end:
  InfixOperatorSymbol: begin if StillCorrect then
       lFormatNr \leftarrow qFormatsColl.CollectFuncForm(qConstructorNr, qLeftLociNbr, qParamNbr);
    gPattern \leftarrow new(InfixFunctorPatternPtr, Init(gPatternPos, gLeftLoci, gConstructorNr, gParams));
    end:
  othercases
         qPattern \leftarrow new(InfixFunctorPatternPtr, Init(qPatternPos, qLeftLoci, qConstructorNr, qParams));
  endcases;
  end:
        The Parser's Read Visible procedure begins by invoking this Start Visible method. The Read Visible
procedure occurs when getting most patterns.
\langle Extended item implementation 1226\rangle + \equiv
procedure extItemObj.StartVisible;
  begin gParams \leftarrow new(PList, Init(0));
  end:
```

 $gParams \leftarrow \mathbf{nil};$

end:

1325. The Parser iteratively calls its GetVisible (§1602) procedure when ReadVisible arguments in a pattern. The GetVisible procedure in turn invokes this ProcessVisible, which increments the number of parameters, and pushes a new Locus object onto the gParams stack.

```
⟨ Extended item implementation 1226⟩ +≡
procedure extItemObj.ProcessVisible;
begin inc(gParamNbr);
if gParams ≠ nil then gParams↑.Insert(new(LocusPtr, Init(CurPos, GetIdentifier)));
end;
```

1326. Recall a structure definition, when it has ancestors, looks like

```
struct (\langle Ancestors \rangle) \langle Structure-Symbol \rangle \cdots
```

The $\langle Ancestors \rangle$ field is considered the "prefix" to the structure definition. The Parser parses a type (thereby populating the gLastType global variable), then invokes the FinishPrefix method, then iterates if it encounters a comma.

The FinishPrefix method pushes the gLastType global variable to the gStructPrefixes state variable.

```
⟨Extended item implementation 1226⟩ +≡

procedure extItemObj.FinishPrefix;

begin gStructPrefixes.Insert(gLastType);

end;

1327. ⟨Extended item implementation 1226⟩ +≡

procedure extItemObj.ProcessStructureSymbol;

var lFormatNr: integer;

begin gConstructorNr \leftarrow 0; gPatternPos \leftarrow CurPos;

if CurWord.Kind = StructureSymbol then gConstructorNr \leftarrow CurWord.Nr;

lFormatNr \leftarrow gFormatsColl.CollectPrefixForm(`J`, gConstructorNr, 1); gParamNbr \leftarrow 0;
```

1328. When the Parser has just finished parsing the ancestors to a structure, but has not parsed the visible arguments. Then the Parser prepares for reading the visible arguments and then the fields by invoking this method. This initializes the *gStructFields* state variable as well as the *gFieldsNbr* state variable.

```
\langle \, \text{Global variables introduced in parseraddition.pas } 1202 \, \rangle \, + \equiv gFieldsNbr: integer;
```

```
1329. \langle \text{Extended item implementation } 1226 \rangle +\equiv  procedure extItemObj.StartFields; var lFormatNr: integer; begin lFormatNr \leftarrow gFormatsColl.CollectPrefixForm(`L`, gConstructorNr, gParamNbr); in\_AggrPattern \leftarrow true; gStructFields \leftarrow new(PList, Init(0)); gFieldsNbr \leftarrow 0; end;
```

1330. The Parser has just encountered the end structure bracket ("#)") token, so we want to add the format to the *qFormatsColl* dictionary.

```
⟨ Extended item implementation 1226⟩ +≡
procedure extItemObj.FinishFields;
var lFormatNr: integer;
begin lFormatNr ← gFormatsColl.CollectPrefixForm(´G´, gConstructorNr, gFieldsNbr);
end;
```

§1331 Mizar Parser PROCESSING DEFINITIONS 453

1331. Recall that each field-segment looks like

```
\langle Field\text{-}Segment \rangle ::= \langle Selector\text{-}Symbol \rangle \{"," \langle Selector\text{-}Symbol \rangle \} \langle Specification \rangle
Before parsing the field-segment, the StartAggrPattSegment is invoked.
\langle \text{Local variables for parser additions } 1209 \rangle + \equiv
gStructFieldsSegment: PList;
gSqmPos: Position;
1332. \langle Extended item implementation 1226 \rangle + \equiv
procedure extItemObj.StartAggrPattSegment;
  begin gStructFieldsSegment \leftarrow new(Plist, Init(0)); gSgmPos \leftarrow CurPos;
  end;
1333. For each selector-symbol the Parser encounters, it invokes the ProcessField.
\langle Extended item implementation 1226\rangle + \equiv
procedure extItemObj.ProcessField;
  var lFormatNr: integer;
  begin lFormatNr \leftarrow qFormatsColl.CollectPrefixForm(`U`, CurWord.Nr, 1);
  gStructFieldsSegment \uparrow. Insert(new(FieldSymbolPtr, Init(CurPos, CurWord.Nr))); inc(gFieldsNbr);
  end;
         After each field has been parsed, the Parser invokes this method to update the gStructFields will
push a new field segment object onto it.
\langle Extended item implementation 1226\rangle + \equiv
procedure extItemObj.FinishAggrPattSegment;
  \mathbf{begin}\ gStructFields.Insert(new(FieldSegmentPtr,Init(gSgmPos,gStructFieldsSegment,gLastType)));
  end:
```

Subsection 23.2.5. Processing remaining statements

1335. Processing schemes. Most of these methods are used in parsing a scheme block (§1654). It will be useful to examine that function to see where these methods are invoked.

When the Parser starts a new scheme, several state variables need to be reset. The gSchemeIdNr is populated by the GetIdentifier (§1201) procedure, the gSchemeIdPos is assigned the current position, and the gSchemeParams should be allocated to an empty list.

```
\langle Extended item implementation 1226\rangle +\equiv procedure extItemObj.ProcessSchemeName; begin gSchemeIdNr \leftarrow GetIdentifier; <math>gSchemeIdPos \leftarrow CurPos; gSchemeParams \leftarrow new(PList,Init(0)); end;
```

1336. A scheme qualification segment looks like, for predicates:

```
\langle Variable \rangle \ \{ \ "," \ \langle Variable \rangle \ \} \ "["[\langle Type\text{-}Expression\text{-}List \rangle]"]"
```

And for functors:

```
\langle Variable \rangle \{ ", " \langle Variable \rangle \} "(" [\langle Type-Expression-List \rangle] ")"
```

When the comma-separated list of identifiers have all been read, but before either "(" or "[" has been discerned, the Parser invokes StartSchemeQualification.

This will assign the current word kind to gSubItemKind, and then initialize the gTypeList to 4 items.

```
\langle Global variables introduced in parseraddition.pas 1202 \rangle += gTypeList\colon MList;
```

```
1337. \langle \text{Extended item implementation } 1226 \rangle +\equiv  procedure extItemObj.StartSchemeQualification; begin gSubItemKind \leftarrow CurWord.Kind; gTypeList.Init(4); end:
```

1338. After the type-list has been parsed, but before the closing parentheses or bracket has been encountered, the Parser invokes the FinishSchemeQualification method. This assigns the current position to the qSubItemPos.

```
\langle Global variables introduced in parseraddition.pas 1202\rangle +\equiv gSubItemPos: Position;
```

```
1339. \langle \text{Extended item implementation } 1226 \rangle + \equiv procedure extItemObj.FinishSchemeQualification; begin gSubItemPos \leftarrow CurPos end;
```

1340. Starting a scheme segment describes the situation where we are *just about* to start parsing the comma-separated list of identifiers for the scheme parameters. This just assigns the current position to the qSubItemPos, then initializes qSchVarIds to 2 spots.

```
\langle Global variables introduced in parseraddition.pas 1202\rangle += gSchVarIds: MList;
```

```
1341. ⟨Extended item implementation 1226⟩ +≡ procedure extItemObj.StartSchemeSegment; begin gSubItemPos ← CurPos; gSchVarIds.Init(2); end;
```

1342. After parsing the identifier for an entry in the comma-separated list of scheme variables, the Parser invokes ProcessScheme Variable to add the recently parsed identifier to the gSch VarIds state variable.

```
⟨ Extended item implementation 1226⟩ +≡
procedure extItemObj.ProcessSchemeVariable;
begin gSchVarIds.Insert(new(VariablePtr, Init(CurPos, GetIdentifier)));
end;
```

1343. Once the list of scheme variables and their type specification has been parsed, then the Parser invokes the *FinishSchemeSegment* method. This just turns the *gSchVarIds* list into a Predicate segment or a Functor segment, using the type list the Parser just finished parsing.

```
 \begin case \ gSubItemKind \ of \\ sy\_LeftParanthesis: \ begin \ gSchemeParams.Insert(new(FunctorSegmentPtr, Init(gSubItemPos, new(PList, MoveList(gSchVarIds)), new(PList, MoveList(gTypeList)), gLastType))); \\ end; \\ sy\_LeftSquareBracket: \ begin \ gSchemeParams.Insert(new(SchemeSegmentPtr, Init(gSubItemPos, PredicateSegment, new(PList, MoveList(gSchVarIds)), new(PList, MoveList(gTypeList))))); \\ end; \\ endcases; \\ end; \\ \endcases; \\ end; \\ \endcases \\ \e
```

1344. The "scheme thesis" is the formula statement of the scheme. Informally, a scheme looks like:

```
scheme \{\langle Scheme-Parameters \rangle\} \langle Scheme-thesis \rangle  "provided" \langle Scheme-premises \rangle
```

This means the gLastFormula state variable contains the scheme's thesis. But the Parser has not yet started the list of premises. This is when the Parser invokes the FinishSchemeThesis method, which assigns the gLastFormula to gSchemeConclusion, then allocates a new empty list for the gSchemePremises.

```
\langle Extended item implementation 1226\rangle +\equiv procedure extItemObj.FinishSchemeThesis; begin <math>gSchemeConclusion \leftarrow gLastFormula; <math>gSchemePremises \leftarrow new(Plist,Init(0)); end;
```

1345. The premises for a scheme consists of finitely many formulas separated by "and" keywords. The Parser enters into a loop invoking this method *after* parsing the formula but *before* checking the next word is "and" (and iterating loop). We just need to push the formula onto the *gSchemePremises* list.

```
\langle \text{ Extended item implementation } 1226 \rangle + \equiv \\ \textbf{procedure } extItemObj.FinishSchemePremise; \\ \textbf{begin } gSchemePremises \uparrow .Insert(new(PropositionPtr, Init(nLabel, gLastFormula, nPropPos))); \\ \textbf{end;} \\ \end{cases}
```

1346. Reserved variables. These methods are invoked only when the Parser parses a reservation (§1651). A "reservation segment" refers to the comma-separated list of variables and the type.

Starting a reservation segment allocates a new (empty) list for gResIdents, and assigns the gResPos to the current position. Each variable encountered in the comma-separated list of variables is appended to the gResIdents list using the ProcessReservedIdentifier method.

Mizar treats each reservation segment as a separate statement. So there is no difference between:

```
reserve G for Group, x,y,z for Element of G;
...and...
reserve G for Group;
reserve x,y,z for Element of G;
```

Finishing a reservation mutates both the gLastWSItem and gLastWSBlock global variables. Specifically, we allocate a new reservation Item, then update gLastWSItem to point to it. The caller's nLastWSItem is updated to point to it, too. We assign the content of this newly allocated reservation Item based on the gResIdents list. We insert this Item to the end of the gLastWSBlock's items.

```
\langle Global variables introduced in parseraddition.pas 1202 \rangle + \equiv
qResIdents: PList;
qResPos: Position;
1347. \langle Extended item implementation 1226 \rangle + \equiv
procedure extItemObj.StartReservationSegment;
  begin gResIdents \leftarrow new(Plist, Init(0)); gResPos \leftarrow CurPos;
  end;
procedure extItemObj.ProcessReservedIdentifier;
  begin qResIdents \uparrow .Insert(new(VariablePtr, Init(CurPos, GetIdentifier)));
  end:
procedure extItemObj.FinishReservationSegment;
  begin qLastWSItem \leftarrow qWsTextProper \uparrow.NewItem(itReservation, qResPos);
  nLastWSItem \leftarrow gLastWSItem;
  qLastWSItem \uparrow .nContent \leftarrow new(ReservationSegmentPtr, Init(qResIdents, qLastType));
  gLastWSItem \uparrow .nItemEndPos \leftarrow PrevPos; \ gLastWSBlock \uparrow .nItems.Insert(gLastWSItem);
  end;
```

1348. Both "defpred" and "deffunc" invokes StartPrivateDefiniendum to initialize the gTypeList, store the identifier in the gPrivateId, and assign the current position to the gPrivateIdPos. Further, $dol_Allowed$ is toggled to true — placeholder variables are going to be allowed in the type declarations of the private functor or private predicate (for example "defpred Foo[set, Element of \$1]").

```
functor or private predicate (for example "defpred Foo[set, Element of $1]").

\( \text{Global variables introduced in parseraddition.pas 1202} \rangle += \frac{gPrivateId: Integer;}{gPrivateIdPos: Position;}

\( \text{Extended item implementation 1226} \rangle += \text{procedure } \text{extItemObj.StartPrivateDefiniendum;} \)

\( \text{begin } \text{gPrivateId} \rightarrow \text{GetIdentifier;} \text{gPrivateIdPos} \rightarrow \text{CurPos;} \text{dol_Allowed} \rightarrow true; \text{gTypeList.Init(4);} \)

\( \text{end;} \)
```

1350. Reading a "type list" (for scheme parameters or for private definitions) loops over reading a type, then pushing it onto the gTypeList. The parser delegates that latter "push work" to the FinishLocusType method.

```
⟨ Extended item implementation 1226⟩ +=
procedure extItemObj.FinishLocusType;
begin gTypeList.Insert(gLastType);
end;
```

1351. The life-cycle of expressions is a little convoluted. The *Item* will allocate a new extExpression object and assign it to the gExpPtr. Later, almost always, the gExpPtr will invoke a method to create a subexpression. This subexpression will be populated, then the gLastTerm (or gLastFormula) will be updated to point to this subexpression object. The expression object will be freed.

```
\langle Extended item implementation 1226\rangle +\equiv procedure extItemObj.CreateExpression(fExpKind: ExpKind); begin <math>gExpPtr \leftarrow new(extExpressionPtr, Init(fExpKind)); end:
```

1352. Recall the "set" statement is of the form

```
"set" \langle Variable \rangle "=" \langle Term \rangle { "," \langle Variable \rangle "=" \langle Term \rangle }
```

The Parser parses this as a loop of assignments of terms to identifiers. Before iterating, the Parser invokes the FinishPrivateConstant method. This allocates a new item for the constant definition, then assigns it to the gLastWSItem and to the caller's nLastWSItem field. Then the content for the new item is allocated to be a constant definition object using the VariablePtr state variable and the gLastTerm state variable. The gLastBlock global variable pushes the new constant definition item to its contents.

```
\langle \text{ Extended item implementation } 1226 \rangle + \equiv \\ \textbf{procedure } extItemObj.FinishPrivateConstant; \\ \textbf{begin } gLastWSItem \leftarrow gWsTextProper \uparrow.NewItem(itConstantDefinition, nItemPos); \\ nLastWSItem \leftarrow gLastWSItem; gLastWSItem \uparrow.nContent \leftarrow new(ConstantDefinitionPtr, Init(new(VariablePtr, Init(gPrivateIdPos, gPrivateId)), gLastTerm)); \\ gLastWSItem \uparrow.nItemEndPos \leftarrow PrevPos; gLastWSBlock \uparrow.nItems.Insert(gLastWSItem); \\ nItemPos \leftarrow CurPos; \\ \textbf{end}; \\ \end{cases}
```

1353. When the Parser is about to start parsing an assignment " $\langle Variable \rangle = \langle Term \rangle$ " in a "set" statement, the Parser invokes this method. The caller assigns the gPrivateId state variable to be the result of GetIdentifier, and the gPrivateIdPos state variable to be the current position.

```
⟨ Extended item implementation 1226⟩ +≡
procedure extItemObj.StartPrivateConstant;
begin gPrivateId ← GetIdentifier; gPrivateIdPos ← CurPos;
end;
```

1354. For a "defpred" and a "deffunc", before parsing the definiens, we need to set the *dol_Allowed* global variable to true (to allow placeholder variables).

```
\langle Extended item implementation 1226\rangle +\equiv procedure extItemObj.StartPrivateDefiniens; begin dol_Allowed \leftarrow true; end;
```

```
After parsing the definiendum term for a "deffunc", the Parser invokes this FinishPrivateFuncDefinienition
method. This assigns the contents of the caller to a WSM private functor definition syntax tree.
\langle Extended item implementation 1226\rangle + \equiv
procedure extItemObj.FinishPrivateFuncDefinienition;
            begin nLastWSItem \uparrow .nContent \leftarrow new(PrivateFunctorDefinitionPtr, Init(new(VariablePtr, Init(new(VariablePt
                                   Init(qPrivateIdPos, qPrivateId)), new(PList, MoveList(qTypeList)), qLastTerm));
            end;
1356. When finishing the definiendum formula for a "defpred", the Parser invokes this FinishPrivatePredDefinienition
method.
\langle Extended item implementation 1226\rangle + \equiv
procedure extItemObj.FinishPrivatePredDefinienition;
            begin nLastWSItem \uparrow .nContent \leftarrow new(PrivatePredicateDefinitionPtr, Init(new(VariablePtr, Init(new(Variable
                                   Init(gPrivateIdPos, gPrivateId)), new(PList, MoveList(gTypeList)), gLastFormula));
            end;
1357.
                                    Reconsider statements.
\langle Extended item implementation 1226\rangle + \equiv
procedure extItemObj.ProcessReconsideredVariable;
            begin qPrivateId \leftarrow GetIdentifier; <math>qPrivateIdPos \leftarrow CurPos;
procedure extItemObj. FinishReconsideredTerm;
            begin gReconsiderList \uparrow. Insert(new(TypeChangePtr, Init(Equating, new(VariablePtr, new(Variable
                                   Init(gPrivateIdPos, gPrivateId)), gLastTerm)));
            end:
1358. This is invoked when parsing a private item which is a "reconsider" statement.
\langle Extended item implementation 1226\rangle + \equiv
procedure extItemObj.FinishDefaultTerm;
            \textbf{begin} \ gReconsiderList \uparrow. Insert (new (\textit{TypeChangePtr}, Init (\textit{VariableIdentifier}, new (\textit{VariablePtr}, n
                                   Init(gPrivateIdPos, gPrivateId)), nil)));
            end:
1359. When the Parser finishes parsing a formula in "consider (Segment) such that (Formula) {and
\langle Formula \rangle \}", the Parser invokes the FinishCondition method. This checks that gPremises has been allo-
cated, then pushes a new labeled formula into it.
\langle Extended item implementation 1226\rangle + \equiv
procedure extItemObj.FinishCondition;
            begin if gPremises = \mathbf{nil} then gPremises \leftarrow new(PList, Init(0));
            qPremises \uparrow. Insert(new(PropositionPtr, Init(nLabel, qLastFormula, nPropPos)));
            end:
```

1360. In statements of the form

```
assume \langle Formula \rangle;
```

Or of the form

```
assume \langle Formula \rangle and \langle Formula \rangle and ... and \langle Formula \rangle;
```

After each formula parsed, the Parser invokes the FinishHypothesis. This just inserts a new labeled formula into the gPremises state variable, when the gPremises state variable is not **nil**.

```
\langle Extended item implementation 1226\rangle +\equiv procedure extItemObj.FinishHypothesis; begin if gPremises \neq nil then gPremises \uparrow.Insert(new(PropositionPtr,Init(nLabel,gLastFormula,nPropPos))); end;
```

1361. "Take" statements. For statements of the form

```
take \langle Variable \rangle = \langle Term \rangle;
```

The Parser invokes the ProcessExemplifyingVariable method, then parses the term, and then constructs the AST by invoking FinishExemplifyingVariable.

Finishing a "take" statement mutates both the gLastWSItem and the gLastWSBlock global variables.

```
\langle Extended item implementation 1226\rangle + \equiv
```

```
procedure extItemObj.ProcessExemplifyingVariable;
```

```
begin gPrivateId \leftarrow GetIdentifier; <math>gPrivateIdPos \leftarrow CurPos; end:
```

```
procedure extItemObj.FinishExemplifyingVariable;
```

```
 \begin{array}{l} \textbf{begin} \ \ gLastWSItem \leftarrow gWsTextProper \uparrow. NewItem(itExemplification, nItemPos); \\ nLastWSItem \leftarrow gLastWSItem; \ gLastWSItem \uparrow. nContent \leftarrow new(ExamplePtr, Init(new(VariablePtr, Init(gPrivateIdPos, gPrivateId)), gLastTerm)); \ gLastWSItem \uparrow. nItemEndPos \leftarrow PrevPos; \\ gLastWSBlock \uparrow. nItems. Insert(gLastWSItem); \ nItemPos \leftarrow CurPos; \\ \textbf{end}; \\ \end{array}
```

```
1362. In statements of the form
```

```
take \langle Term \rangle;
the Parser begins by invoking StartExemplifyingTerm, parses the term, then FinishExemplifyingTerm.
\langle Extended item implementation 1226\rangle + \equiv
procedure extItemObj.StartExemplifyingTerm;
             begin if (CurWord.Kind = Identifier) \land extBlockPtr(gBlockPtr) \uparrow .nInDiffuse \land ((AheadWord.Kind = Identifier)) \land (Identifier) \land (Identifie
                                                     sy\_Comma) \vee (AheadWord.Kind = sy\_Semicolon)) then
                          begin qPrivateId \leftarrow GetIdentifier; <math>qPrivateIdPos \leftarrow CurPos;
             else gPrivateId \leftarrow 0;
             end;
procedure extItemObj.FinishExemplifyingTerm;
             begin gLastWSItem \leftarrow gWsTextProper \uparrow.NewItem(itExemplification, nItemPos);
             nLastWSItem \leftarrow gLastWSItem;
             if gPrivateId \neq 0 then gLastWSItem \uparrow .nContent \leftarrow new(ExamplePtr, Init(new(VariablePtr, Init(new(VariablePt
                                                     Init(gPrivateIdPos, gPrivateId)), nil))
             else gLastWSItem \uparrow .nContent \leftarrow new(ExamplePtr, Init(nil, gLastTerm));
             qLastWSItem \uparrow .nItemEndPos \leftarrow PrevPos; \ qLastWSBlock \uparrow .nItems.Insert(qLastWSItem);
             nItemPos \leftarrow CurPos;
             end;
```

1363. When the Parser examines the correctness conditions (§1622), it loops over the correctness conditions and justifications. Afterwards, it invokes the ProcessCorrectness method, which tests that the parser is not current looking at a correctness keyword. Then it tests if gCorrectnessConditions is empty or AxiomsAllowed (in which case, correctness has been satisfies, so the Parser moves happily along). But if $gCorrectnessConditions \neq \emptyset$ or axioms are not allowed, then a 73 error is raised.

```
\langle Extended item implementation 1226\rangle +\equiv procedure extItemObj.ProcessCorrectness; begin if CurWord.Kind \neq sy\_Correctness then if (gCorrectnessConditions \neq []) \land \neg AxiomsAllowed then Error(gDefPos, 73); end;
```

1364. A "construction type" appears in a redefinition where the type is redefined. In such a situation, we need to add "coherence" as a correctness condition. The *StartConstructionType* handles this task.

```
\langle Extended item implementation 1226\rangle += procedure extItemObj.StartConstructionType; begin if gRedefinitions \wedge (CurWord.Kind = sy\_Arrow) then include(gCorrectnessConditions, syCoherence); end;
```

1365. This is used in the Parser's ProcessLab procedure. Really, all the work is being done here: the nLabel field of the caller is assigned to a newly allocated Label object.

end:

end;

end:

1366. A regular statement is either a "diffuse" statement (which occurs with the "now" keyword) or else it's a "compact" statement.

(Extended item implementation 1226) $+\equiv$ procedure extItemObj.StartRegularStatement;

begin if $CurWord.Kind = sy_Now$ then $nRegularStatementKind \leftarrow stDiffuseStatement$ else $nRegularStatementKind \leftarrow stCompactStatement$;

1367. If the Parser encounters a colon after the copula, then it invokes this method to construct a label for the Definiens.

```
\langle Global variables introduced in parseraddition.pas 1202\rangle +\equiv gDefLabel: LabelPtr;

1368. \langle Extended item implementation 1226\rangle +\equiv procedure extItemObj.ProcessDefiniensLabel; begin gDefLabId \leftarrow 0; gDefLabPos \leftarrow CurPos; if (CurWord.Kind = Identifier) \wedge (AheadWord.Kind = sy\_Colon) then gDefLabId \leftarrow CurWord.Nr; gDefLabel \leftarrow new(LabelPtr, Init(gDefLabId, gDefLabPos));
```

1369. The Parser, having encountered "from" and a non-MML reference, tries to treat the identifier as the label for a scheme declared in the current article. The nInference field would be a SchemeJustification object, so we just populate its nSchemeIdNr and position fields.

```
⟨ Extended item implementation 1226⟩ +≡
procedure extItemObj.ProcessSchemeReference;
begin if CurWord.Kind = Identifier then
   begin SchemeJustificationPtr(nInference)↑.nSchemeIdNr ← CurWord.Nr;
   SchemeJustificationPtr(nInference)↑.nSchemeInfPos ← CurPos;
   end;
end;
```

1370. When a "by" refers to a theorem or definition from an article in the MML, the Parser invokes the *StartLibraryReference* method.

```
    ⟨Global variables introduced in parseraddition.pas 1202⟩ +≡ gTHEFileNr: integer;
    1371. ⟨Extended item implementation 1226⟩ +≡ procedure extItemObj.StartLibraryReferences;
    begin gTHEFileNr ← CurWord.Nr;
```

1372. The Parser has already encountered a "from" and then an MML article identifier. Before continuing to parse the scheme number, the Parser invokes this method to initialize the relevant state variables.

```
\langle Extended item implementation 1226\rangle += procedure extItemObj.StartSchemeLibraryReference; begin gTHEFileNr \leftarrow CurWord.Nr; end;
```

end; end;

1373. For references to labels found in the article being processed ("private references"), this method is invoked. \langle Extended item implementation 1226 $\rangle + \equiv$ **procedure** extItemObj.ProcessPrivateReference; **begin** $Simple Justification Ptr(nInference) \uparrow .nReferences \uparrow .Insert(new(Local Reference Ptr, new Local Reference)) \(\)$ Init(GetIdentifier, CurPos))); end; 1374. When using a definition from an MML article in a scheme reference (something like "from MyScheme(ARTICLE:def 5,...)"), well, the Parser stores this fact in a state variable gDefinitional. The *ProcessDef* method populates this state variable correctly. \langle Global variables introduced in parseraddition.pas $1202 \rangle + \equiv$ *qDefinitional*: boolean; 1375. \langle Extended item implementation $1226\rangle + \equiv$ **procedure** *extItemObj.ProcessDef*; **begin** $gDefinitional \leftarrow (CurWord.Kind = ReferenceSort) \land (CurWord.Nr = ord(syDef))$ end; When accumulating the references in a Scheme-Justification, and a reference is from an MML article, 1376.ProcessTheoremNumber transforms it into a newly allocated reference object. The caller's nInference then adds the newly allocated object to its nReferences collection. \langle Extended item implementation 1226 $\rangle + \equiv$ **procedure** extItemObj.ProcessTheoremNumber; var lRefPtr: ReferencePtr; **begin if** $CurWord.Kind \neq Numeral$ **then** exit; if CurWord.Nr = 0 then **begin** ErrImm(146); exitend: **if** qDefinitional **then** $lRefPtr \leftarrow new(DefinitionReferencePtr, Init(<math>qTHEFileNr, CurWord.Nr, CurPos)$) else $lRefPtr \leftarrow new(TheoremReferencePtr, Init(gTHEFileNr, CurWord.Nr, CurPos));$ $Simple Justification Ptr(nInference) \uparrow .nReferences \uparrow .Insert(lRefPtr);$ end: When a Scheme-Justification uses a local reference, the Parser delegates the work to the *Item*'s ProcessSchemeNumber method. This updates the caller's nInference field. \langle Extended item implementation 1226 $\rangle + \equiv$ **procedure** extItemObj.ProcessSchemeNumber; **begin if** $CurWord.Kind \neq Numeral$ **then** exit; if CurWord.Nr = 0 then **begin** ErrImm(146); exitend: with $SchemeJustificationPtr(nInference) \uparrow do$

 $\textbf{begin} \ \textit{nSchFileNr} \leftarrow \textit{gTHEFileNr}; \ \textit{nSchemeIdNr} \leftarrow \textit{CurWord.Nr}; \ \textit{nSchemeInfPos} \leftarrow \textit{PrevPos};$

1378. This appears when the Parser starts its *Justification* ($\S1578$) procedure, or in the *RegularStatement* ($\S1599$) procedure.

This clears the *nInference*, reassigning it to the **nil** pointer.

 \langle Extended item implementation 1226 $\rangle + \equiv$

end:

end;

procedure *extItemObj.FinishSimpleJustification*; **begin with** *extBlockPtr(gBlockPtr)*↑ **do**

 $extBlockPtr(gBlockPtr)\uparrow.nLinked \leftarrow false;$

begin if $is_inference_error$ **then** $nInference \uparrow .nInfSort \leftarrow infError$;

For nested "proof" blocks, check if the 'check proofs' ("::\$P+") pragma has been enabled — if so, just set the caller's *nInference* to be a new Justification object with a 'proof' tag. Otherwise, we're skipping the proofs, so set *nInference* to be the 'skipped' justification.

```
\langle Extended item implementation 1226\rangle + \equiv
procedure extItemObj.StartJustification;
  begin nInference \leftarrow nil;
  if CurWord.Kind = sy\_Proof then
    begin if ProofPragma then nInference \leftarrow new(JustificationPtr, Init(infProof, CurPos))
    else nInference \leftarrow new(JustificationPtr, Init(infSkippedProof, CurPos))
    end;
  end;
1379. A simple justification is either a Scheme-Justification ("from..."), a Straightforward-Justification
("by..."), or... somethign else?
\langle Extended item implementation 1226\rangle + \equiv
procedure extItemObj.StartSimpleJustification;
  begin case CurWord.Kind of
  sy-From: nInference \leftarrow new(SchemeJustificationPtr, Init(CurPos, 0, 0));
  sy_By: with extBlockPtr(gBlockPtr)\uparrow do
       nInference \leftarrow new(StraightforwardJustificationPtr, Init(CurPos, nLinked, nLinkPos));
  othercases with extBlockPtr(gBlockPtr)\uparrow do
       nInference \leftarrow new(StraightforwardJustificationPtr, Init(PrevPos, nLinked, nLinkPos));
  endcases;
  end;
1380. We should update the nInference field's sort to be infError when, well, the inference is an error
(e.g., the Parser is in panic mode). We should set the qBlockPtr's nLinked field to false when we just added
a straightforward justification (or an erroneous justification).
  define is\_inference\_error \equiv \neg StillCorrect \lor
              ((CurWord.Kind \neq sy\_Semicolon) \land (CurWord.Kind \neq sy\_DotEquals)) \lor
              ((nInference \uparrow .nInfSort = infStraightforwardJustification) \land (byte(nLinked) >
              byte(nLinkAllowed))) \lor ((nInference \uparrow .nInfSort = infSchemeJustification) \land
              (SchemeJustificationPtr(nInference)\uparrow.nSchemeIdNr = 0))
```

if $(nInference \uparrow .nInfSort = infStraightforwardJustification) \lor (nInference \uparrow .nInfSort = infError)$ then

1381. For iterative equalities, we should recall that it looks like

```
LHS = RHS \langle Justification \rangle
.= RHS2
.= ...;
```

This matters because, well, when the Parser has parsed "LHS = RHS $\langle Justification \rangle$ ", the Parser believes it is a compact statement. Until the Parser looks at the next token, it does not know whether this is a Compact-Statement or an iterated equality. The FinishCompactMethod peeks at the token, and when the token is an iterated equality (".=") updates the caller's fields as well as initialize the gIterativeLastFormula, gIterativeSteps, and gInference state variables. The gBlockPtr is updated to make its nLinked field false.

```
\langle Extended item implementation 1226\rangle +\equiv procedure extItemObj.FinishCompactStatement; begin if CurWord.Kind = sy\_DotEquals then begin gIterativeLastFormula \leftarrow gLastFormula; nRegularStatementKind \leftarrow stIterativeEquality; <math>extBlockPtr(gBlockPtr)\uparrow.nLinked \leftarrow false; gIterativeSteps \leftarrow new(PList,Init(0)); gInference \leftarrow nInference; end; end;
```

1382. Every time the Parser encounters the ".=" token, it immediately invokes the *StartIterativeStep* method. This just updates the *gIterPos* state variable to the current position.

```
\langle Local variables for parser additions 1209\rangle +\equiv gIterPos: Position;
```

1383. $\langle \text{Extended item implementation } 1226 \rangle + \equiv$ **procedure** extItemObj.StartIterativeStep; **begin** $gIterPos \leftarrow CurPos;$ **end**;

1384. Right before the Parser iterates the loop checking if ".=" is the next token for an iterative equation, the Parser invokes the FinishIterativeStep method. This just adds a new IterativeStep object, an AST node representing the preceding ".= RHS by $\langle Justification \rangle$ ".

```
⟨ Extended item implementation 1226⟩ +≡
procedure extItemObj.FinishIterativeStep;
begin gIterativeSteps↑.Insert(new(IterativeStepPtr, Init(gIterPos, gLastTerm, nInference)));
end:
```

1385. In a definition, after the Parser finishes parsing the definiens, we construct the AST node for it with the *FinishDefiniens* method.

For each copula ("means" and "equals"), the algorithm is the same: if we just had a definition-by-cases, then store the "otherwise" clause in *lExp* and assign the *gDefiniens* state variable to a newly allocated conditional definiens object. If the definiens is not a definition-by-cases (i.e., it's a "simple" definition), then just assign *gDefiniens* a newly allocated *SimpleDefiniens* object.

For functor definitions (not redefinitions), the gCorrectnessConditions are assigned here.

```
\langle Extended item implementation 1226\rangle + \equiv
procedure extItemObj.FinishDefiniens;
  var lExp: DefExpressionPtr:
  begin case gDefiningWay of
  dfMeans:
    if gPartialDefs \neq nil then
       begin lExp \leftarrow nil;
       if qOtherwise \neq nil then lExp \leftarrow new(DefExpressionPtr, Init(exFormula, qOtherwise));
       qDefiniens \leftarrow new(ConditionalDefiniensPtr, Init(qMeansPos, qDefLabel, qPartialDefs, lExp))
    else gDefiniens \leftarrow new(SimpleDefiniensPtr, Init(gMeansPos, gDefLabel, new(DefExpressionPtr,
            Init(exFormula, gLastFormula))));
  dfEquals:
    if qPartialDefs \neq nil then
       begin lExp \leftarrow nil;
       if qOtherwise \neq nil then lExp \leftarrow new(DefExpressionPtr, Init(exTerm, qOtherwise));
       gDefiniens \leftarrow new(ConditionalDefiniensPtr, Init(gMeansPos, gDefLabel, gPartialDefs, lExp))
    else gDefiniens \leftarrow new(SimpleDefiniensPtr, Init(qMeansPos, gDefLabel, new(DefExpressionPtr,
            Init(exTerm, gLastTerm))));
  endcases;
  if \neg gRedefinitions \land (nItemKind = itDefFunc) then
    begin if gDefiningWay = dfMeans then gCorrectnessConditions \leftarrow [syExistence, syUniqueness]
    else if gDefiningWay = dfEquals then gCorrectnessConditions \leftarrow [syCoherence];
    end;
  end;
```

Section 23.3. EXTENDED SUBEXPRESSION CLASS

1386. Aside: refactoring. We should probably refactor a private procedure *PushTermStack* to push a new term onto the term stack, and a private function *PopTermStack* to return the top of the term stack (and mutate the term stack), and possibly a *ResetTermStack* procedure (which will clear the term stack and possibly the objects stored in it?).

We see that TermNbr is decremented when popping the Term stack (via FinishTerm); when FinishQualifyingFormula is invoked, it decrements the TermNbr; when FinishAttributiveFormula is invoked, it decrements the TermNbr; but these latter two methods can (and should) be refactored to use the FinishTerm to pop the term stack and decrement the TermNbr state variable.

Assigning the TermNbr occurs when CreateArgs method is invoked; the InsertIncorrBasic method resets the TermNbr to the nTermBase; the ProcessAtomicFormula, when a 157 error is raised, will reset the TermNbr to the nTermBase; when the constructor for an extExpression object is invoked, it resets the TermNbr to zero (which happens in the extItem's CreateExpression method—which occurs frequently enough to be a worry).

The only time when the TermNbr is incremented is when we push a new term onto the Term stack.

1387. There is a comment in Polish "teraz jest to kolekcja MultipleTypeExp", which Google translates to "now it is a MultipleTypeExp collection". I have made this replacement in the code below, prefixed with a "+" sign (to distinguish it from the other comment already in English).

Also note: the nRestriction refers to the subformula in a universally quantified formula

```
1388.
        \langle Extended subexpression class declaration 1388\rangle \equiv
  extSubexpPtr = \uparrow extSubexpObj;
  extSubexpObj = \mathbf{object} (SubexpObj)
    nTermBase, nRightArgBase: integer;
    nSubexpPos, nNotPos, nRestrPos: Position;
    nQuaPos: Position;
    nSpelling: Integer;
    nSymbolNr, nRSymbolNr: integer;
    nConnective, nNextWord: TokenKind;
    nModeKind: TokenKind;
    nModeNr: integer;
    nRightSideOfPredPos: Position;
    nMultipredicateList: MList;
    nSample: TermPtr; \{ for Fraenkel terms \}
    nAllPos: Position;
    nPostQualList: MList;  { + now it is a MultipleTypeExp collection }
    nQualifiedSegments: MList;
    nSegmentIdentColl: MList; { quantified variables, keeps spellings of vars }
    nSegmentPos: Position;
    nFirstSententialOperand: FormulaPtr;
    nRestriction: FormulaPtr;
    nAttrCollection: MList;
    nNoneOcc: boolean;
    nNonPos: Position;
    nPostNegated: boolean;
    nArgListNbr: integer; { position in a term (§1519) }
    nArgList: array of arg_type;
    nFunc: array of func_type;
    constructor Init;
    \langle Methods implemented by subclasses of SubexpObj 716\rangle
  end;
This code is used in section 1199.
1389. The TermNbr is used to treat a list of terms as a stack data structure. Specifically, the Term array
is treated as a stack, and the TermNbr is the index of the "top" of the stack.
\langle \text{Local variables for parser additions } 1209 \rangle + \equiv
TermNbr: integer;
```

```
468
                                                                                                                                                                                                    Mizar Parser
1390.
                   \langle Extended subexpression implementation 1390\rangle \equiv
          { Subexpressions handling }
constructor extSubexpObj.Init;
     const MaxArqListNbr = 20;
     begin inherited Init; nRestriction \leftarrow nil; nTermBase \leftarrow TermNbr; nArgListNbr \leftarrow 0;
     setlength(nArgList, MaxArgListNbr + 1); setlength(nFunc, MaxArgListNbr + 1);
     nArgList[0].Start \leftarrow TermNbr + 1;
     end:
See also sections 1391, 1392, 1393, 1395, 1396, 1397, 1398, 1399, 1400, 1401, 1402, 1403, 1404, 1405, 1406, 1407, 1408, 1409,
          1410, 1411, 1413, 1417, 1429, 1430, 1431, 1432, 1433, 1434, 1435, 1436, 1437, 1438, 1439, 1440, 1441, 1442, 1443, 1444,
          1445,\ 1446,\ 1447,\ 1448,\ 1449,\ 1450,\ 1451,\ 1452,\ 1453,\ 1454,\ 1455,\ 1456,\ 1457,\ 1458,\ 1459,\ 1460,\ 1461,\ 1462,\ 1463,\ 1464,\ 1461,\ 1462,\ 1463,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 1464,\ 
          1465, 1466, 1467, 1468, 1469, 1470, 1471, 1472, 1473, 1474, 1475, 1476, 1477, 1478, 1479, 1480, 1481, 1482, 1483, 1484,
          1485, 1486, 1487, and 1488.
This code is used in section 1200.
1391. When the Parser is about to parse a stack of attributes, either in a registration or on a type, we
need to initialize the appropriate state variables. We also need the caller's nAttrCollection to be initialized
with an empty list.
\langle Extended subexpression implementation 1390\rangle + \equiv
procedure extSubexpObj.StartAttributes;
     begin nAttrCollection.Init(0); gLastType \leftarrow nil;
1392. When the Parser expects an adjective, and the caller is used to store the adjective or attribute, we
need to check if it is negated. This handles it.
\langle Extended subexpression implementation 1390\rangle + \equiv
procedure extSubexpObj.ProcessNon;
     begin nNoneOcc \leftarrow CurWord.Kind = sy\_Non; nNonPos \leftarrow CurPos;
     end;
1393. Pop arguments from term stack. This will take some parameter aBase and copy pointers to
```

each element of Term[aBase .. TermNbr] into a new list. Then the TermNbr state variable is updated to be aBase - 1.

This means that executing "list1 \leftarrow CreateArgs(aBase); list2 \leftarrow CreateArgs(aBase);" will have list2 = nil.

Bug: when aBase < 0, this will set TermNbr to a negative number.

```
\langle Extended subexpression implementation 1390\rangle + \equiv
function CreateArgs(aBase: integer): PList;
  var k: integer; lList: PList;
  begin lList \leftarrow new(PList, Init(TermNbr - aBase));
  for k \leftarrow aBase to TermNbr do lList.Insert(Term[k]);
  TermNbr \leftarrow aBase - 1; CreateArgs \leftarrow lList;
  end:
```

1394. The "process (singular) attribute" method is invoked in the "process (plural) attributes" procedure ($\S1524$), and in the ATTSubexpression procedure ($\S1641$). This method will be invoked when the Parser is looking at an attribute token.

When there is no format recorded for such an attribute, then a 175 error will be raised.

This will allocate a new Adjective object, store it in the gLastAdjective state variable, then append it to the nAttrCollection field of the caller.

```
\langle Global variables introduced in parseraddition.pas 1202\rangle + \equiv
gLastAdjective: AdjectiveExpressionPtr;
```

procedure extSubexpObj.CompleteAdjectiveCluster;

end:

begin $gAttrColl \leftarrow new(PList, MoveList(nAttrCollection));$

```
1395.
        \langle Extended subexpression implementation 1390 \rangle + \equiv
procedure extSubexpObj.ProcessAttribute;
  var lFormatNr: integer;
  begin if CurWord.Kind = AttributeSymbol then
    begin
         lFormatNr \leftarrow gFormatsColl.LookUp\_PrefixFormat(`V`, CurWord.Nr, TermNbr - nTermBase + 1);
    if lFormatNr = 0 then { format not found! }
       begin qLastAdjective \leftarrow new(AdjectivePtr, Init(CurPos, 0, CreateArgs(nTermBase + 1)));
       Error(CurPos, 175)
       end
    else begin
           qLastAdjective \leftarrow new(AdjectivePtr, Init(CurPos, CurWord.Nr, CreateArgs(nTermBase + 1)));
      if nNoneOcc then qLastAdjective \leftarrow new(NegatedAdjectivePtr, Init(nNonPos, qLastAdjective));
      end:
    end
          { needed for ATTSubexpression adjective cluster handling }
  begin gLastAdjective \leftarrow new(AdjectivePtr, Init(CurPos, 0, CreateArgs(nTermBase + 1)));
  nAttrCollection.Insert(qLastAdjective);
  end:
1396.
        These next next method is invoked before the Parser parses arguments for an attribute.
\langle Extended subexpression implementation 1390\rangle + \equiv
procedure extSubexpObj.StartAttributeArguments;
  begin nTermBase \leftarrow TermNbr;
  end;
1397.
        The next two methods are invoked after the Parser has finished parsing the arguments for an
attribute.
  I am confused why there is duplicate code here, and the naming conventions suggest the FinishAttributeArguments
method should be preferred.
\langle Extended subexpression implementation 1390 \rangle + \equiv
procedure extSubexpObj.CompleteAttributeArguments;
  begin nSubexpPos \leftarrow CurPos; nRightArgBase \leftarrow TermNbr;
  end;
procedure extSubexpObj.FinishAttributeArguments;
  begin nSubexpPos \leftarrow CurPos; nRightArgBase \leftarrow TermNbr;
  end:
        This allocates a new list of pointers, moves the caller's nAttrCollection into the list, and updates
the qAttrColl state variable to point at them.
  Again, this should be named FinishedAdjectiveCluster to be consistent with the naming conventions
seemingly adopted.
\langle Extended subexpression implementation 1390 \rangle + \equiv
```

1399. When the Parser works its way through a registration block, check that the TermNbr points to not farther ahead than one more token ahead from the caller's nTermBase field. Raise an error if that happens. This method is only invoked in the Parser module's the RegisterCluster (§1643) procedure.

```
⟨ Extended subexpression implementation 1390⟩ +≡
procedure extSubexpObj.CompleteClusterTerm;
begin if TermNbr - nTermBase > 1 then
   begin ErrImm(379); gLastTerm ← new(IncorrectTermPtr, Init(CurPos));
   end;
end;
```

1400. A "simple term" appears to be a variable. This is used when the Parser parses an identifier as a closed term ($\S1509$). The state variable *gLastTerm* is updated to point to a newly allocated *SimpleTerm* object.

This method should probably be moved closer to the other methods used when parsing terms.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.ProcessSimpleTerm; begin gLastTerm \leftarrow new(SimpleTermPtr, Init(CurPos, GetIdentifier)); end;
```

1401. Qualified terms. The Parser invokes ProcessQua when it is looking directly at a "qua" token, specifically in the AppendQua (§1504) procedure. The ProcessQua method is used nowhere else. It is solely responsible for "marking the current position" of the Parser, and storing that in the caller's nQuaPos field.

```
⟨ Extended subexpression implementation 1390⟩ +≡ procedure extSubexpObj.ProcessQua; begin nQuaPos ← CurPos end;
```

1402. The Parser invokes the *FinishedQualifiedTerm* method after encountering a "qua" and after parsing the type. This method constructs a new *QualifiedTerm* object reflecting the top of the *Term* stack is taken "qua" the *gLastType*, and the mutates the top of the *Term* stack to be this newly allocated *QualifiedTerm* object.

This method does not push anything new to the term stack, but it does mutate the Term stack. This method is used nowhere else other than the Parser's AppendQua (§1504) procedure.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.FinishQualifiedTerm; begin <math>Term[TermNbr] \leftarrow new(QualifiedTermPtr, Init(nQuaPos, Term[TermNbr], gLastType)); end;
```

1403. Although the "exactly" reserved keyword is not used for anything, the method for *ProcessExactly* marks the current position and stores it in the caller's nQuaPos, then updates (**not** pushes) to the top of the term stack by turning the top of the stack into an ExactlyTerm object.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.ProcessExactly; begin nQuaPos \leftarrow CurPos; Term[TermNbr] \leftarrow new(ExactlyTermPtr, Init(nQuaPos, Term[TermNbr])); end;
```

1404. Arguments to a term. The *CheckTermLimit* procedure is a "private helper function" for the *FinishArgument* method.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure CheckTermLimit; 
var l: integer; 
begin if TermNbr \geq length(Term) then 
begin l \leftarrow 2 * length(Term); setlength(Term, l); 
end; 
end;
```

1405. Pushing the Term stack. This method pushes the *gLastTerm* state variable's contents to the *Term* stack, mutating the *TermNbr* and *Term* module-local variables.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.FinishArgument; begin CheckTermLimit; inc(TermNbr); Term[TermNbr] \leftarrow gLastTerm; end;
```

1406. Pop the Term stack. The evil twin to "pushing" an element onto a stack, "popping" a stack removes the top element. We pop the *Term* stack whenever we finish the term.

```
This is only used in AppendFunc (§1519).
```

This should probably check that the *Term* stack is not empty before being invoked.

```
\langle \text{ Extended subexpression implementation } 1390 \rangle + \equiv
procedure extSubexpObj.FinishTerm;
begin gLastTerm \leftarrow Term[TermNbr]; dec(TermNbr);
end;
```

Subsection 23.3.1. Parsing Types

1407. When we start parsing a new type, we make sure the gLastType state variable is not caching an old type. We assign it to be the **nil** pointer.

```
⟨ Extended subexpression implementation 1390⟩ +≡
procedure extSubexpObj.StartType;
begin gLastType ← nil;
end;
```

1408. This is invoked only by the Parser's RadixTypeSubexpression (§1525) procedure. The Parser delegates the work of storing the mode information to this method. In turn, the caller's nModeKind field stores the current word's token Kind, and the caller's nModeNr field stores the current word's number. The Parser's current position is marked and stored in the caller's nSubexpPos field.

But no state variables are mutated by this method.

```
\langle \text{ Extended subexpression implementation } 1390 \rangle + \equiv \\ \textbf{procedure } extSubexpObj.ProcessModeSymbol; \\ \textbf{begin } nModeKind \leftarrow CurWord.Kind; nModeNr \leftarrow CurWord.Nr; \\ \textbf{if } (CurWord.Kind = sy\_Set) \quad \{? \land (AheadWord.Kind \neq sy\_Of)?\} \\ \textbf{then } nModeKind \leftarrow ModeSymbol; nSubexpPos \leftarrow CurPos; \\ \textbf{end :} \\ \end{cases}
```

472 Parsing types Mizer Parser $\S1409$

1409. The Parser has just finished parsing a type and its arguments — " $\langle Mode \rangle$ of $\langle Term-list \rangle$ " or " $\langle Structure \rangle$ over $\langle Term-list \rangle$ ". The data has been accumulated into the caller, which will now be constructed into an AST object. The newly allocated AST node will be stored in the gLastType state variable.

If the caller is trying to construct a mode which does not match the format recorded in the gFormatsColl, a 151 error will be raised.

Similarly, if the caller is trying to construct a structure which does not match the format recorded in the *qFormatsColl*, a 185 error will be raised.

This is invoked only by the Parser's RadixTypeSubexpression (§1525) procedure.

```
\langle Extended subexpression implementation 1390\rangle + \equiv
procedure extSubexpObj.FinishType:
  var lFormatNr: integer;
  begin case nModeKind of
  ModeSymbol: begin
         lFormatNr \leftarrow gFormatsColl.LookUp\_PrefixFormat(`M`, nModeNr, TermNbr - nTermBase);
    if lFormatNr = 0 then Error(nSubexpPos, 151); {format missing}
    qLastType \leftarrow new(StandardTypePtr, Init(nSubexpPos, nModeNr, CreateArgs(nTermBase + 1)));
    end;
  StructureSymbol: begin
         lFormatNr \leftarrow gFormatsColl.LookUp\_PrefixFormat(`L`, nModeNr, TermNbr - nTermBase);
    if lFormatNr = 0 then SemErr(185); { format missing }
    qLastType \leftarrow new(StructTypePtr, Init(nSubexpPos, nModeNr, CreateArgs(nTermBase + 1)));
  othercases begin qLastType \leftarrow new(IncorrectTypePtr, Init(CurPos)); end;
  endcases;
  end;
```

1410. If the Parser has the misfortune of trying to make sense of a malformed type expression, then with a heavy heart it invokes this method to update the gLastType state variable to be an incorrect type expression at the current position.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.InsertIncorrType; begin gLastType \leftarrow new(IncorrectTypePtr, Init(CurPos)); end;
```

 $\S1411$ Mizar Parser PARSING TYPES 473

1411. When the parser encounters a qualifying formula (" $\langle Term \rangle$ is $\langle Type \rangle$ ") or is parsing a type for a cluster (the "cluster ...for $\langle Type \rangle$ "), after parsing the type, this method is invoked to **update** the gLastType state variable to store the ClusteredType AST node (which decorates a type — the contents of gLastType at the time of calling — with a bunch of attributes).

The caller's nAttrCollection is transferred to the gLastType. At the end of the method, the caller's nAttrCollection (array of pointers) is freed. This does not free the objects referenced by the pointers, however.

If gLastType = nil, then the Parser has somehow failed to parse the type expression. An error should be raised.

```
\langle \text{ Extended subexpression implementation } 1390 \rangle + \equiv \\ \textbf{procedure } extSubexpObj.CompleteType; \\ \textbf{var } j: integer; \\ \textbf{begin } mizassert(5433, gLastType \neq \textbf{nil}); \\ \textbf{if } nAttrCollection.Count > 0 \textbf{ then} \\ \textbf{begin } gLastType \leftarrow new(ClusteredTypePtr, Init(gLastType\uparrow.nTypePos, new(PList, Init(nAttrCollection.Count)), gLastType)); \\ \textbf{for } j \leftarrow 0 \textbf{ to } nAttrCollection.Count - 1 \textbf{ do} \\ ClusteredTypePtr(gLastType)\uparrow.nAdjectiveCluster\uparrow.Insert(PObject(nAttrCollection.Items\uparrow[j])); \\ nAttrCollection.DeleteAll; \\ \textbf{end}; \\ \textbf{end}; \\ \textbf{end}; \\ \end{aligned}
```

Subsection 23.3.2. Parsing operator precedence

1412. Mario Carneiro's "Mizar in Rust" ($\S6.2$) gives an overview of this parsing routine (see also his mizar-rs/src/parser/miz.rs for the Rust version of the same code). It is a constrained optimization problem. We shall take care to dissect this routine. This appears to be where operator precedence, the gPriority ($\S648$) global variable, comes into play.

1413. Starting a "long term".

We can observe that nTermBase is initialized upon construction to TermNbr; in ProcessAtomicFormula and StartPrivateFormula it is assigned to TermNbr.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.StartLongTerm; begin nArgListNbr \leftarrow 0; nArgList[0].Length \leftarrow TermNbr - nTermBase; end;
```

1414. Malformed term errors.

We should remind the reader, errors 165–175 are "unknown functor format", errors 176 is "unknown attribute format", and error 177 is "unknown structure format". Only when such an error occurs, the flow experiences a **goto** AfterBalance.

For an example of a 168, 169 error:

```
for x being Nat
holds (id + x +) = x;
```

For an example of a 170, 171 error (the first 0 will be flagged 170, the second 0 will be flagged as 171):

```
for x being Nat
holds 0 0 + x = x;
```

For an example of a 172, 173 error:

```
for x being Nat holds x + / = x;
```

For an example of a 174, 175 error:

```
for x being Nat holds x + (1,2) + x = x;
```

1415. We can recall that a "generic" term looks like an infixed operator of the form

$$(t_1^{(\ell)}, \dots, t_m^{(\ell)}) t (t_1^{(r)}, \dots, t_n^{(r)})$$

The parentheses are optional. Constants will have m=n=0 and look like () t (). Function-like terms will have m=0 and look like () t ($t_1^{(r)}, \ldots, t_n^{(r)}$). The problem statement could be re-phrased as: given several infixed terms without parentheses inserted anywhere, determine how to cluster terms together.

1416. The problem statement for constructing the syntax tree for a term is something like the following: we have an expression of the form

$$x_1^{(0)}, \ldots, x_{k_0}^{(0)} F_1 x_1^{(1)}, \ldots, x_{k_1}^{(1)} F_2 \cdots F_n x_1^{(n)}, \ldots, x_{k_n}^{(n)}$$

We want to produce a suitable binary tree with F_i on the internal nodes and the $(x_j^{(i)})_{j \leq k_i}$ on the leafs, respecting precedence such that each F_i is applied to the correct number of arguments.

Mario Carneiro noted (arXiv:2304.08391, §6.2) the existence of an $O(n^4)$ algorithm using dynamic programming techniques. The trick is to compute the minimal "cost" [number of violations] for each substring of nodes $F_a \cdots F_b$ for each $1 \le a \le i \le b \le n$ with node F_i being the root of the subtree. There are $O(n^3)$ such subproblems, and they can be calculated from smaller subproblems in O(n). This might seem alarmingly large, but usually the terms in Mizar are sufficiently small.

It is interesting to see how other languages tackle this problem, so I am going to give a haphazard literature review:

- (1) Nils Anders Danielsson and Ulf Norell's "Parsing Mixfix Operators" (in SB. Scholz and O. Chitil (eds.), Symposium on Implementation and Application of Functional Languages, Springer 2008, pp. 80–99; doi:10.1007/978-3-642-24452-0_5) discuss how Agda approaches parsing mixfix operators with different precedence.
- (2) The Isabelle proof assistant uses a modified version of Earley parsing of terms, supporting precedence between 0 to 1000.

1417. The only two place where FinishLongTerm is invoked is in the AppendFunc procedure (§1519) in parser.pas.

This relies on MFormatsList.LookUpFuncFormat (§667), which attempts to look up an MInfixFormatObj (§655) with a given id number as well as number of left and right arguments.

We will need to populate ArgsLength and To_Right to determine the syntax tree for the term (which is our real goal here). The ArgsLength encodes the number of terms are to the left and right of each "internal node". The To_Right controls associativity (which is how Mizar handles operator precedence): if node F_{k+1} is higher precedence than node F_k , then $To_Right(k)$ is true.

The Exchange(i) procedure will make node i a child of i-1 (when node i is a child of i-1), and vice-versa. Visually, this means we transform the tree as:

$$(\cdots F_{i-1} x_1, \ldots, x_\ell), x_{\ell+1}, \ldots, x_n F_i \cdots \longleftrightarrow \cdots F_{i-1} x_1, \ldots, x_{\ell-1}, (x_\ell, x_{\ell+1}, \ldots, x_n F_i \cdots)$$

Observe that "Exchange(i); Exchange(i)" is equivalent to doing nothing.

We should recall ($\S1387$) that nArgList is an array of "**record** Instance, SymPri: integer; FuncPos: Position; end".

```
\langle Extended subexpression implementation 1390\rangle + \equiv
procedure extSubexpObj.FinishLongTerm;
  var ArgsLength: array of record l, r: integer;
       end;
     To_Right: array of boolean;
    procedure Exchange(i:integer);
       var l: integer;
       \mathbf{begin}\ l \leftarrow ArgsLength[i].l;\ ArgsLength[i].l \leftarrow ArgsLength[i-1].r;\ ArgsLength[i-1].r \leftarrow l;
       To\_Right[i-1] \leftarrow \neg To\_Right[i-1];
       end;
  var Bl, new\_Bl: integer; \{indexes nFunc, ArgsLength\}
    i, j, k: integer; { various indices }
     (Variables for finishing a long term in a subexpression 1426)
  label Corrected, AfterBalance;
  begin (Rebalance the long term tree 1418)
AfterBalance: (Construct the term's syntax tree after balancing arguments among subterms 1427)
  end:
```

1418. Rebalancing the term tree.

setlength(Depo, nArqListNbr + 1);

See also section 1422.

This code is used in section 1418.

```
Note that nArgListNbr is mutated only in extSubexpObj.ProcessFunctorSymbol (§1429), and in ProcessAtomicFormula
(\S1459) it is reset to zero.
  define missing\_functor\_format \equiv qFormatsColl.LookUp\_FuncFormat(Instance, l, r) = 0
\langle Rebalance the long term tree 1418\rangle \equiv
  (Initialize To_Right and ArgsLength arrays 1421)
  (Initialize Bl, goto AfterBalance if term has at most one argument 1423)
    \{Bl = 1 \lor Bl = 2\}
  for k \leftarrow 2 to nArgListNbr - 1 do
    with nFunc[k], ArgsLength[k] do
       begin if missing\_functor\_format then \langle Guess the k^{th} functor format 1424\rangle
     Corrected: end:
  for j \leftarrow nArgListNbr downto Bl + 1 do
    with nFunc[j], ArgsLength[j] do
       begin if \neg missing\_functor\_format then goto AfterBalance;
       Exchange(j); (Check for 172/173 error, goto AfterBalance if erred 1419)
  (Check for 174/175 error, goto AfterBalance if erred 1420)
This code is used in section 1417.
1419. (Check for 172/173 error, goto AfterBalance if erred 1419) \equiv
  if missing_functor_format then
    begin Error(FuncPos, 172); Error(nFunc[nArgListNbr].FuncPos, 173); goto AfterBalance; end;
This code is used in section 1418.
        \langle Check for 174/175 error, goto AfterBalance if erred 1420\rangle \equiv
  with nFunc[Bl], ArgsLength[Bl] do
    if missing_functor_format then
       begin Error(FuncPos, 174); Error(nFunc[nArgListNbr].FuncPos, 175); goto AfterBalance; end;
This code is used in section 1418.
1421. We first allocate the arrays, the we initialize the values.
\langle \text{Initialize } To\_Right \text{ and } ArgsLength \text{ arrays } 1421 \rangle \equiv
  setlength(ArgsLength, nArgListNbr + 1); setlength(To\_Right, nArgListNbr + 1);
```

1422. The initial guess depends on whether F_k has precedence over F_{k+1} or not.

If F_{k+1} has higher precedence than F_k , then the initial guess groups terms as:

$$\cdots F_k \left((x_1^{(k)}, \dots, x_{m_k}^{(k)}) F_{k+1}(\cdots) \right) \cdots$$
, and $To_Right[k] = true$.

On the other hand, if F_{k+1} does not have higher precedence than F_k , then we guess the terms are grouped as

$$\cdots \left(\cdots F_k(x_1^{(k)},\ldots,x_{m_k}^{(k)})\right) \ F_{k+1}\cdots, \quad \text{and} \quad \textit{To_Right}[k] = \textit{false}\,.$$

This is a first stab, but sometimes we get lucky and it's correct.

```
define next\_term\_has\_higher\_precedence(\#) \equiv gPriority.Value(ord(`O`), nFunc[\#].Instance) < gPriority.Value(ord(`O`), nFunc[\#+1].Instance)
\langle \text{Initialize } To\_Right \text{ and } ArgsLength \text{ arrays } 1421 \rangle + \equiv ArgsLength[1].l \leftarrow nArgList[0].Length; To\_Right[0] \leftarrow true;
\text{for } k \leftarrow 1 \text{ to } nArgListNbr - 1 \text{ do}
\text{with } ArgsLength[k] \text{ do}
\text{if } next\_term\_has\_higher\_precedence(k) \text{ then}
\text{begin } r \leftarrow 1; ArgsLength[k+1].l \leftarrow nArgList[k].Length; To\_Right[k] \leftarrow true \text{ end}
\text{else begin } r \leftarrow nArgList[k].Length; ArgsLength[k+1].l \leftarrow 1; To\_Right[k] \leftarrow false \text{ end};
ArgsLength[nArgListNbr].r \leftarrow nArgList[nArgListNbr].Length; To\_Right[nArgListNbr] \leftarrow false;
```

1423. The first situation we encounter is if the user tries to tell Mizar to evaluate something like:

```
for x being Nat holds x + (1,2) = x;
```

Mizar will not understand "x + (1,2)" because it is an invalid functor format — the format would look something like $\langle "+", \text{left} : 1, \text{right} : 1 \rangle$ but the format of the expression is $\langle \text{left} : 1, \text{right} : 2 \rangle$. The mismatch on the "right" values in the formats will raise a 165 error.

For a 166 error example,

```
for x being Nat
holds + / = x;
```

Mizar will not like the leading "+ /" expression, and flag this with the 166 error.

Mizar will flag "+ 0" as a 165 error.

```
⟨Initialize Bl, goto AfterBalance if term has at most one argument 1423⟩ ≡ with nFunc[1], ArgsLength[1] do
begin if nArgListNbr = 1 then
begin if missing\_functor\_format then
begin Error(FuncPos, 165); goto AfterBalance end;
goto AfterBalance;
end;
Bl \leftarrow 1;
if missing\_functor\_format then
begin Exchange(2); Bl \leftarrow 2;
if missing\_functor\_format then
begin Error(FuncPos, 166); goto AfterBalance end;
end;
```

This code is used in section 1418.

478

```
\langle \text{Guess the } k^{th} \text{ functor format } 1424 \rangle \equiv
1424.
  begin Exchange(k+1); new\_Bl \leftarrow Bl;
  if missing_functor_format then
    begin if Bl = k then
       begin Error(nFunc[k-1].FuncPos, 168); Error(FuncPos, 169); goto AfterBalance; end;
    Exchange(k+1); Exchange(k); new\_Bl \leftarrow k;
    if missing_functor_format then
       begin Exchange(k+1); new\_Bl \leftarrow k+1;
       if missing_functor_format then
         begin Error (FuncPos, 167); goto AfterBalance end;
       end;
    for j \leftarrow k-1 downto Bl+1 do
       with nFunc[j], ArgsLength[j] do
         begin if ¬missing_functor_format then goto Corrected;
         Exchange(j);
         if missing_functor_format then
            begin Error(FuncPos, 168); Error(nFunc[k].FuncPos, 169); goto AfterBalance; end;
    \langle Check term Bl has valid functor format, goto AfterBalance if not 1425\rangle
    end:
  Bl \leftarrow new\_Bl;
  end:
This code is used in section 1418.
        \langle Check term Bl has valid functor format, goto AfterBalance if not 1425 \rangle \equiv
  with nFunc[Bl], ArgsLength[Bl] do
    if missing_functor_format then
       begin Error(FuncPos, 170); Error(nFunc[k].FuncPos, 171); goto AfterBalance; end;
This code is used in section 1424.
1426. Constructing the syntax tree. The second half of finishing a long term constructs the syntax
tree for the term.
\langle \text{Variables for finishing a long term in a subexpression } 1426 \rangle \equiv
ak, pl, ll, kn: integer;
lTrm: TermPtr;
lLeftArgs, lRightArgs: PList;
DepoNbr: integer:
Depo: array of record FuncInstNr: integer;
    dArgList: PList;
    end;
This code is used in section 1417.
```

```
1427.
         \langle Construct the term's syntax tree after balancing arguments among subterms 1427 \rangle \equiv
  \langle Initialize symbol priorities, determine last ll, pl values 1428\rangle
  DepoNbr \leftarrow 0:
  for kn \leftarrow nArqListNbr downto 2 do
     if To_Right[kn-1] then { if kn node is parent of kn-1 node }
       begin with nFunc[kn] do
          begin lRightArgs \leftarrow CreateArgs(nArgList[kn].Start); \{ (\S1393) \}
          lLeftArgs \leftarrow CreateArgs(nArgList[kn-1].Start);
          lTrm \leftarrow new(InfixTermPtr, Init(FuncPos, Instance, lLeftArgs, lRightArgs));
          end:
       for j \leftarrow DepoNbr downto 1 do
          with Depo[j], nFunc[FuncInstNr] do
            begin if symPri \leq nFunc[kn-1].SymPri then break;
            dec(DepoNbr); lLeftArgs \leftarrow new(PList, Init(1)); lLeftArgs \uparrow .Insert(lTrm);
            lTrm \leftarrow new(InfixTermPtr, Init(FuncPos, Instance, lLeftArgs, dArgList));
            end;
       gLastTerm \leftarrow lTrm;
       qSubexpPtr \uparrow . FinishArgument;
       end
     else begin inc(DepoNbr);
       with Depo[DepoNbr] do
          begin FuncInstNr \leftarrow kn; dArgList \leftarrow CreateArgs(nArgList[kn].Start); end;
       end;
  with nFunc[1] do
     begin lRightArgs \leftarrow CreateArgs(nArgList[1].Start); lLeftArgs \leftarrow CreateArgs(nArgList[0].Start);
     lTrm \leftarrow new(InfixTermPtr, Init(FuncPos, Instance, lLeftArgs, lRightArgs));
     end:
  for j \leftarrow DepoNbr downto 1 do
     with Depo[j], nFunc[FuncInstNr] do
       begin lLeftArgs \leftarrow new(PList, Init(1)); lLeftArgs \uparrow. Insert(lTrm);
       lTrm \leftarrow new(InfixTermPtr, Init(FuncPos, Instance, lLeftArgs, dArgList));
       end:
  qLastTerm \leftarrow lTrm;
This code is used in section 1417.
1428. (Initialize symbol priorities, determine last ll, pl values 1428) \equiv
  for ak \leftarrow 1 to nArgListNbr do
     begin ll \leftarrow 1; pl \leftarrow 1;
    if To\_Right[ak-1] then ll \leftarrow nArgList[ak-1].Length;
    if \neg To\_Right[ak] then pl \leftarrow nArgList[ak].Length;
     with nFunc[ak] do
       begin symPri \leftarrow qPriority. Value(ord(`O`), Instance); end;
     end;
This code is used in section 1427.
```

Subsection 23.3.3. Processing subexpressions

1429. Note that ProcessFunctorSymbol is the only place where nArgListNbr is incremented. Processing functor symbols occurs in the parser's AppendFunc (§1519) in a loop.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.ProcessFunctorSymbol; var l: integer; begin inc(nArgListNbr); if nArgListNbr \geq length(nFunc) then begin l \leftarrow 2 * length(nFunc) + 1; setlength(nArgList,l); setlength(nFunc,l); end; nArgList[nArgListNbr].Start \leftarrow TermNbr + 1; nFunc[nArgListNbr].FuncPos \leftarrow CurPos; nFunc[nArgListNbr].Instance \leftarrow CurWord.Nr; end;
```

1430. The Parser is in the middle of AppendFunc and has just finished parsing a term t or a tuple of terms (t_1, \ldots, t_n). Before the Parser checks if it's looking at an infixed functor operator or not, the Parser invokes the FinishArgList method. It's the only time where the FinishArgList method is invoked.

This allocates either 1 or n to the length of nArgList[nArgListNbr], to store the information for the term(s).

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.FinishArgList; begin nArgList[nArgListNbr].Length \leftarrow TermNbr - nArgList[nArgListNbr].Start + 1; end;
```

1431. The Parser is looking at "where" or (when the variables are all reserved) a colon ":", the Parser invokes the *StartFraenkelTerm* which will store the previous term in the *nSample* field — so schematically, the Fraenkel term could look like

```
 \left\{ \langle nSample \rangle \text{ where } \langle Postqualification \rangle : \langle Formula \rangle \right\}   \left\langle \text{ Extended subexpression implementation } 1390 \right\rangle + \equiv   \mathbf{procedure } \ extSubexpObj.StartFraenkelTerm;   \mathbf{begin } \ nSample \leftarrow gLastTerm;   \mathbf{end};
```

1432. This is only invoked in the Parser's *ProcessPostqualification* (§1506) procedure, which is only invoked after the Parser calls the *extSubexp* object's *StartFraenkelTerm* method.

```
⟨ Extended subexpression implementation 1390⟩ +≡
procedure extSubexpObj.StartPostqualification;
begin nPostQualList.Init(0);
end;
```

1433. The parser is looking at the post-qualified segment of a Fraenkel operator. This will be a list of variables "being" a type, we allocate an array for the variables. This is handled by the *StartPostQualifyingSegment* method.

```
⟨ Extended subexpression implementation 1390⟩ +≡
procedure extSubexpObj.StartPostQualifyingSegment;
begin nSegmentIdentColl.Init(2);
end;
```

1434. While looping over the comma-separated list of variables in a post-qualified segment (in a Fraenkel term), the Parser invokes the *ProcessPostqualifiedVariable* on each iteration until it has parsed all the variables. This allocates a new *Variable* object, and pushes it onto the *nSegmentIdentColl* "stack".

```
⟨ Extended subexpression implementation 1390⟩ +≡
procedure extSubexpObj.ProcessPostqualifiedVariable;
begin nSegmentIdentColl.Insert(new(VariablePtr,Init(CurPos,GetIdentifier)));
end;
```

1435. The parser is looking at "is" or "are" in a Fraenkel term's post-qualification segment, but has not yet parsed the type. This method will assign the *nSegmentPos* field to be the current position, and assign the *gLastType* state variable to be the **nil** pointer.

```
⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure extSubexpObj.StartPostqualificationSpecyfication;
begin nSegmentPos ← CurPos; gLastType ← nil;
end;
```

1436. The Parser has just parsed either (1) a comma-separated list of variables, the copula "is" or "are", and the type; or (2) a comma-separated list of reserved variables (but no copula and no type). We just need to construct an appropriate node for the abstract syntax tree. This method will append a new Segment to the *nPostQualList*.

```
\langle Extended subexpression implementation 1390\rangle + \equiv
procedure extSubexpObj.FinishPostQualifyingSegment;
  var k: integer; lSegment: ExplicitlyQualifiedSegmentPtr;
  begin if gLastType \neq nil then
     \mathbf{begin}\ lSegment \leftarrow new(ExplicitlyQualifiedSegmentPtr, Init(nSegmentPos, new(PList, Init(0)),
          gLastType); nPostQualList.Insert(lSegment);
     for k \leftarrow 0 to nSegmentIdentColl.Count - 1 do
       begin Explicitly Qualified Segment Ptr(lSegment) \uparrow . nIdentifiers . Insert(nSegment Ident Coll . Items <math>\uparrow [k]);
       end;
     end
  else begin for k \leftarrow 0 to nSegmentIdentColl.Count - 1 do
       begin nPostQualList.Insert(new(ImplicitlyQualifiedSegmentPtr,
            Init(VariablePtr(nSegmentIdentColl.Items\uparrow[k])\uparrow.nVarPos,nSegmentIdentColl.Items\uparrow[k]));
       end;
     end;
  nSegmentIdentColl.DeleteAll; nSegmentIdentColl.Done;
  end;
```

1437. The Parser has just finished the formula in a Fraenkel term, and it is staring at the closet "}" bracket. The Parser invokes this method to construct a new FraenkelTerm AST node, and updates the gLastTerm to point at it.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.FinishFraenkelTerm; begin gLastTerm \leftarrow new(FraenkelTermPtr, Init(CurPos, new(PList, MoveList(nPostQualList)), <math>nSample, gLastFormula)); end;
```

1438. The Parser has already encountered "the set" and the next token is "of", which means the Parser has encountered a "simple" Fraenkel term of the form "the set of all $\langle Term \rangle$...". This method will be invoked once the Parser has stumbled across the "all". The caller updates its nAllPos to the Parser's current position.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.StartSimpleFraenkelTerm; begin <math>nAllPos \leftarrow CurPos; end;
```

1439. The Parser has just finished parsing the post-qualification to the simple Fraenkel term, which means it has finished parsing the simple Fraenkel term. This method allocates a new *SimpleFraenkelTerm* AST node with the accumulated AST nodes, then updates the *gLastTerm* to point to the allocated *SimpleFraenkelTerm* node.

```
 \langle \text{ Extended subexpression implementation } 1390 \rangle + \equiv \\ \textbf{procedure } extSubexpObj.FinishSimpleFraenkelTerm; \\ \textbf{begin } gLastTerm \leftarrow new(SimpleFraenkelTermPtr, Init(nAllPos, new(PList, MoveList(nPostQualList)), nSample)); \\ \textbf{end;}
```

1440. The Parser is looking at a closed term of the form " $\langle Identifier \rangle$ ", and so it looks like a private functor. This method updates the caller's nSubexpPos to the Parser's current position, and the nSpelling is assigned to the identifier's number (for the private functor).

```
⟨ Extended subexpression implementation 1390⟩ +≡
procedure extSubexpObj.StartPrivateTerm;
begin nSubexpPos ← CurPos; nSpelling ← CurWord.Nr;
end;
```

1441. The Parser just finished parsing all the arguments to the private functor, and is looking at the closing parentheses for the private functor. This method allocates a new PrivateFunctorTerm object, using the arguments just parsed, and updates the gLastTerm state variable to point to it.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.FinishPrivateTerm; 

begin <math>gLastTerm \leftarrow new(PrivateFunctorTermPtr, Init(nSubexpPos, nSpelling, CreateArgs(nTermBase + 1))); 

end;
```

1442. The Parser has just encountered either a left bracket term or the opening left bracket for a set " $\{$ ". The Parser calls this method, which just updates the caller's nSymbolNr to be whatever the current token's numeric ID value is.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.StartBracketedTerm; begin nSymbolNr \leftarrow CurWord.Nr; end;
```

1443. If the Parser is in panic mode, this method does nothing.

Either the Parser has finished parsing an enumerated set $\{x_1, \ldots, x_n\}$ or a bracketed term. We need to double check the format for the bracket matches what is stored in the *gFormatsColl*, and raise a 152 error if there's a mismatch. Otherwise, allocate a new AST node for the bracketed term, and use CreateArgs on the terms contained within the brackets.

```
 \langle \text{ Extended subexpression implementation } 1390 \rangle + \equiv \\ \textbf{procedure } extSubexpObj.FinishBracketedTerm; \\ \textbf{var } lFormatNr: integer; \\ \textbf{begin if } StillCorrect \textbf{ then} \\ \textbf{begin } nRSymbolNr \leftarrow CurWord.Nr; \ lFormatNr \leftarrow gFormatsColl.LookUp\_BracketFormat(nSymbolNr, nRSymbolNr, TermNbr - nTermBase, 0, 0); \\ \textbf{if } lFormatNr = 0 \textbf{ then } SemErr(152); \\ gLastTerm \leftarrow new(CircumfixTermPtr, Init(CurPos, nSymbolNr, nRSymbolNr, CreateArgs(nTermBase + 1))); \\ \textbf{end;} \\ \textbf{end;} \\ \textbf{end;}
```

1444. Remember that Mizar calls "an instance of structure" an "**Aggregate**". When the Parser is parsing for a closed subterm and has stumbled across a structure constructor (§1510), it first invokes this method. This stores the ID number for the structure in the caller's nSymbolNr.

```
⟨ Extended subexpression implementation 1390⟩ +≡ procedure extSubexpObj.StartAggregateTerm; begin nSymbolNr ← CurWord.Nr; end;
```

1445. The Parser has just parsed the arguments for the structure constructor, and the Parser is now looking at the "#)" token. This method is invoked.

We should check the format for the structure constructor is stored in the gFormatsColl. If not, raise a 176 error. Otherwise, we allocate a new AggregateTerm with the parsed arguments, and then update the gLastTerm pointer to point at it.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.FinishAggregateTerm; 

var <math>lFormatNr: integer; begin lFormatNr \leftarrow gFormatsColl.LookUp\_PrefixFormat(`G`, nSymbolNr, TermNbr - nTermBase); 

if <math>lFormatNr = 0 then Error(CurPos, 176); { missing format error } 

gLastTerm \leftarrow new(AggregateTermPtr, Init(CurPos, nSymbolNr, CreateArgs(nTermBase + 1))); 

end;
```

1446. The Parser is parsing for a closed subterm, and has stumbled across "the" and is looking at a selector token ($\S1515$). This method is invoked. We assign the caller's nSymbolNr to the ID number for the selector token, assign the caller's nSubexpPos to the Parser's current position, and store the next token's kind (i.e., the "of" token's kind) in the nNextWord field.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.StartSelectorTerm; begin <math>nSymbolNr \leftarrow CurWord.Nr; nSubexpPos \leftarrow CurPos; nNextWord \leftarrow AheadWord.Kind; end;
```

1447. The Parser has just parsed "the $\langle Selector \rangle$ of $\langle Term \rangle$ ". Now this method is invoked to assemble the parsed data into an AST node.

If there is no selector with this matching format, then a 182 error will be raised.

If the caller's nNextWord is an "of" token's kind, then we're describing a selector term. We update the gLastTerm state variable to point to a newly allocated SelectorTerm object with the appropriate data set.

On the other hand, "internal selectors" occur when defining a structure. For example,

```
struct (1-sorted) multMagma (#
  carrier -> set,
  multF -> BinOp of the carrier
#);
```

Observe the multF specification is BinOp of the carrier. That "the carrier" is an internal selector. In this case, allocate a new *InternalSelectorTerm* object, and update the *gLastTerm* state variable to point to it.

If, for some reason, the Parser is in neither situation, then just gLastTerm state variable to be an incorrect term.

```
\langle \text{ Extended subexpression implementation } 1390 \rangle + \equiv \\ \textbf{procedure } extSubexpObj.FinishSelectorTerm; \\ \textbf{var } lFormatNr: integer; \\ \textbf{begin } lFormatNr \leftarrow gFormatsColl.LookUp\_PrefixFormat(`U`, nSymbolNr, 1); \\ \textbf{if } lFormatNr = 0 \textbf{ then } Error(nSubexpPos, 182); \\ \textbf{if } nNextWord = sy\_Of \textbf{ then } \\ gLastTerm \leftarrow new(SelectorTermPtr, Init(nSubexpPos, nSymbolNr, gLastTerm)) \\ \textbf{else } \textbf{if } in\_AggrPattern \textbf{ then } \\ gLastTerm \leftarrow new(InternalSelectorTermPtr, Init(nSubexpPos, nSymbolNr)) \\ \textbf{else } \textbf{begin } gLastTerm \leftarrow new(IncorrectTermPtr, Init(nSubexpPos)); Error(nSubexpPos, 329) \\ \textbf{end; } \\ \textbf{end; } \end{aligned}
```

1448. The Parser is about to start parsing a forgetful functor ($\S1516$) — for example "the multMagma of REAL.TopGroup". This method is invoked. The caller's nSymbolNr field is updated to the current token's ID Number, the nSubexpPos field is assigned the Parser's current position, and the nNextWord field is assigned to the token kind of the next token — this is expected to be "of".

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.StartForgetfulTerm; begin nSymbolNr \leftarrow CurWord.Nr; nSubexpPos \leftarrow CurPos; nNextWord \leftarrow AheadWord.Kind; end:
```

1449. The Parser just finished parsing a forgetful functor. If the Parser is not panicking, check the format for the forgetful functor matches what is stored in the *gFormatsColl* state variable. If the format is invalid, raise a 184 error.

Whether the Parser is panicking or not, allocate a new ForgetfulFunctor term, and update the gLastTerm to point to it.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.FinishForgetfulTerm;
var lFormatNr: integer;
begin lFormatNr \leftarrow 0;
if StillCorrect then
begin lFormatNr \leftarrow gFormatsColl.LookUp\_PrefixFormat(`J`, nSymbolNr, 1);
if lFormatNr = 0 then Error(nSubexpPos, 184); { missing format}
end;
gLastTerm \leftarrow new(ForgetfulFunctorTermPtr, Init(nSubexpPos, nSymbolNr, gLastTerm));
end;
```

1450. There are several situations where this is invoked:

- (1) The Parser has just parsed "the" but is not looking at a selector symbol ("the multF of..."), nor is the Parser looking at a forgetful functor ("the multMagma of..."). Then this is interpreted as looking at a choice operator (§1515).
- (2) The Parser has just parsed "the" but is not looking at a forgetful functor, so the Parser believes it must be looking at a choice operator (§1516).
- (3) The Parser has just parsed "the" and is now looking at "set" so this is invoking the axiom of choice to pick "the set" (§1517).

In these three situations, the Parser invokes this method. It just updates the caller's nSubexpPos field to point to the Parser's current position.

```
⟨ Extended subexpression implementation 1390⟩ +≡
procedure extSubexpObj.StartChoiceTerm;
begin nSubexpPos ← CurPos;
end;
```

1451. The Parser has just parsed a type, and now believes it has finished parsing a choice expression. Then it invokes this method to construct an appropriate AST node for the term, by specifically allocating a new *ChoiceTerm* for the *gLastType* type. We then update the *gLastTerm* state variable to point to this newly allocated term.

```
\langle Extended subexpression implementation 1390 \rangle +\equiv procedure extSubexpObj.FinishChoiceTerm; begin gLastTerm \leftarrow new(ChoiceTermPtr, Init(nSubexpPos, gLastType)); end;
```

1452. When the Parser encounters a numeral while seeking a closed subterm ($\S1508$), it invokes this method to allocate a new *NumeralTerm*. The *gLastTerm* state variable is updated to point to this newly allocated numeral object.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.ProcessNumeralTerm; 
begin <math>gLastTerm \leftarrow new(NumeralTermPtr, Init(CurPos, CurWord.Nr)); end;
```

486

The Parser tries to parse a closed subterm ($\S1508$) and encounters the "it" token. Well, if the it_Allowed state variable is true, then we should allocate a new ItTerm and update the qLastTerm state variable to point to it.

Otherwise, when the it_Allowed state variable is false, we should raise a 251 error.

```
\langle Extended subexpression implementation 1390\rangle + \equiv
procedure extSubexpObj.ProcessItTerm;
  begin if it\_Allowed then gLastTerm \leftarrow new(ItTermPtr, Init(CurPos))
  else begin gLastTerm \leftarrow new(IncorrectTermPtr, Init(CurPos)); ErrImm(251)
    end;
  end:
```

The Parser tries parsing for a closed subterm and has encountered a placeholder term for a private functor (e.g., "\$1"). If the dol_Allowed state variable is true, then allocate a new PlaceholderTerm object and update the *gLastTerm* state variable to point at it.

If the dol_Allowed state variable is false, then we should raise a 181 error.

```
\langle Extended subexpression implementation 1390\rangle + \equiv
procedure extSubexpObj.ProcessLocusTerm;
  begin if dol_Allowed then qLastTerm \leftarrow new(PlaceholderTermPtr, Init(CurPos, CurWord.Nr))
  else begin qLastTerm \leftarrow new(IncorrectTermPtr, Init(CurPos)); ErrImm(181)
    end;
  end;
```

Calamity! An incorrect expression has crossed the Parser's path. Allocate an *IncorrectTerm* object located at the Parser's current position, then update the qLastTerm state variable to point to it.

```
\langle Extended subexpression implementation 1390 \rangle + \equiv
procedure extSubexpObj.InsertIncorrTerm;
  begin gLastTerm \leftarrow new(IncorrectTermPtr, Init(CurPos));
  end:
```

Subsection 23.3.4. Parsing formulas

The Parser is trying to parse an atomic formula (§1544), but something has gone awry. Allocate a new IncorrectFormula object located at the Parser's current position, update the gLastFormula state variable to point to it, and "reset" the TermNbr state variable to point to where the caller's nTermBase is located.

```
\langle Extended subexpression implementation 1390\rangle + \equiv
procedure extSubexpObj.InsertIncorrBasic;
  begin qLastFormula \leftarrow new(IncorrectFormulaPtr, Init(CurPos)); TermNbr \leftarrow nTermBase;
  end;
```

1457. While the Parser was trying to parse a formula, it found something which "doesn't quite fit". Allocate a new IncorrectFormula object, then update the gLastFormula state variable to point to it.

```
\langle Extended subexpression implementation 1390\rangle + \equiv
procedure extSubexpObj.InsertIncorrFormula;
  begin gLastFormula \leftarrow new(IncorrectFormulaPtr, Init(CurPos));
  end:
```

 $\{1458$ Mizar Parser PARSING FORMULAS 487

1458. If we are in a proof, allocate a new *ThesisFormula* object (recall the WEB macro for this §????). Otherwise, raise a 65 error.

```
⟨ Extended subexpression implementation 1390⟩ +≡
procedure extSubexpObj.ProcessThesis;
begin if gProofCnt > 0 then gLastFormula ← thesis_formula
else begin ErrImm(65); gLastFormula ← new(IncorrectFormulaPtr, Init(CurPos));
end;
end;
```

1459. The Parser has encountered " $\langle Term \rangle$ is", or some other generic atomic formula (§1544), this method is invoked.

If more than one term appears before the "is" token (i.e., if $TermNbr - nTermBase \neq 1$), then a 157 error is raised. There is a Polish comment here, "Trzeba chyba wstawic recovery dla TermNbr = nTermBase", which I translated to English.

This will initialize the fields for the caller in preparation for parsing some atomic formula. In particular, this is the only place where TermNbr is initialized to a nonzero value (and isn't in an incorrect formula).

```
\langle \text{ Extended subexpression implementation } 1390 \rangle + \equiv \\ \textbf{procedure } extSubexpObj.ProcessAtomicFormula; \\ \textbf{const } MaxArgListNbr = 20; \\ \textbf{begin } nSubexpPos \leftarrow CurPos; nSymbolNr \leftarrow 0; \\ \textbf{case } CurWord.Kind \ \textbf{of} \\ sy\_Is: \ \textbf{if } TermNbr - nTermBase \neq 1 \ \textbf{then} \\ \textbf{begin } ErrImm(157); \ TermNbr \leftarrow nTermBase; InsertIncorrTerm; FinishArgument; \\ \{ \text{ I think you need to insert recovery for } TermNbr = nTermBase \} \\ \textbf{end;} \\ \textbf{endcases;} \\ nRightArgBase \leftarrow TermNbr; \ nTermBase \leftarrow TermNbr; \ nPostNegated \leftarrow false; \ nArgListNbr \leftarrow 0; \\ nArgList[0].Start \leftarrow TermNbr + 1; \\ \textbf{end;} \end{aligned}
```

1460. The Parser is either finishing a "predicative formula" (§1543) or it's parsing a predicate pattern (§1612), it invokes this method to initialize the fields needed when forming an AST node. Specifically, the nSubexpPos is assigned to the Parser's current position, the nSymbolNr is updated either to the current token's ID number (if the current token is "=" or a predicate) or else assigned to be zero. Last, the nRightArgBase is assigned to equal the TermNbr state variable.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.ProcessPredicateSymbol; begin nSubexpPos \leftarrow CurPos; case CurWord.Kind of sy\_Equal, PredicateSymbol: nSymbolNr \leftarrow CurWord.Nr; othercases nSymbolNr \leftarrow 0; endcases; nRightArgBase \leftarrow TermNbr; end:
```

488 PARSING FORMULAS Mizar Parser $\S1461$

The Parser is parsing a "predicate formula" which has arguments on the righthand side of the predicate symbol ($\S1539$). \langle Extended subexpression implementation 1390 $\rangle + \equiv$ **procedure** extSubexpObj.ProcessRightSideOfPredicateSymbol; **begin** $nRightSideOfPredPos \leftarrow CurPos$; case CurWord.Kind of sy_Equal , PredicateSymbol: $nSymbolNr \leftarrow CurWord.Nr$; othercases $nSymbolNr \leftarrow 0$; endcases; $nRightArgBase \leftarrow TermNbr;$ end: The Parser has just finished a "predicate formula" (§1543), then this method is invoked to construct 1462. an AST for the formula. First we check if the format is valid. If the format for the formula is not found in the qFormatsColl, then we must raise a 153 error. Otherwise, we construct two lists (one for the left arguments, another for the right arguments), and use them to construct a new *PredicativeFormula* object. We update the gLastFormula state variable to point to the newly allocated formula object. \langle Extended subexpression implementation $1390 \rangle + \equiv$ **procedure** extSubexpObj.FinishPredicativeFormula; var lLeftArgs, lRightArgs: PList; lFormatNr: integer; **begin** $lFormatNr \leftarrow qFormatsColl.LookUp_PredFormat(nSymbolNr, nRightArqBase - nTermBase,$ TermNbr - nRightArgBase); if lFormatNr = 0 then Error(nSubexpPos, 153); { missing format } $lRightArgs \leftarrow CreateArgs(nRightArgBase + 1);\ lLeftArgs \leftarrow CreateArgs(nTermBase + 1);$ $qLastFormula \leftarrow new(PredicativeFormulaPtr, Init(nSubexpPos, nSymbolNr, lLeftArgs, lRightArgs));$ end; 1463. The Parser tries to construct an AST when finishing up the right-hand side of a predicative formula (§1539), it invokes this method after the extSubexpObj.FinishPredicativeFormula has been invoked. \langle Extended subexpression implementation $1390 \rangle + \equiv$ **procedure** extSubexpObj.FinishRightSideOfPredicativeFormula; var lRightArgs: PList; lLeftArgsNbr, lFormatNr: integer; lFrm: FormulaPtr; **begin** $lFrm \leftarrow gLastFormula$; if $lFrm \uparrow .nFormulaSort = wsNegatedFormula$ then $lFrm \leftarrow NegativeFormulaPtr(lFrm) \uparrow .nArq$; $lLeftArgsNbr \leftarrow RightSideOfPredicativeFormulaPtr(lFrm)\uparrow.nRightArgs\uparrow.Count;$ $lFormatNr \leftarrow gFormatsColl.LookUp_PredFormat(nSymbolNr, lLeftArgsNbr, TermNbr - nRightArgBase);$ if lFormatNr = 0 then Error(nSubexpPos, 153); { missing format } $lRightArgs \leftarrow CreateArgs(nRightArgBase + 1);$ $qLastFormula \leftarrow new(RightSideOfPredicativeFormulaPtr, Init(nSubexpPos, nSymbolNr, lRightArgs));$ nMultiPredicateList.Insert(gLastFormula);end; When the Parser is parsing an atomic formula, when it has parsed a formula and encounters another predicate, it defaults to thinking that it is starting a "multi-predicative formula" (§1540), and it invokes this method. This initializes the nMultiPredicateList to an empty list of length 4, and the first entry points to the same formula pointed to by the gLastFormula state variable. \langle Extended subexpression implementation 1390 $\rangle + \equiv$ **procedure** extSubexpObj.StartMultiPredicativeFormula; **begin** nMultiPredicateList.Init(4); nMultiPredicateList.Insert(qLastFormula);

end;

§1465 Mizar Parser PARSING FORMULAS 489

1465. Finishing a "multi-predicative formula" allocates a new *MultiPredicativeFormula* object, and moves the contents of the caller's *nMultiPredicateList* to the newly minted formula. The *gLastFormula* state variable is updated to point to this newly allocated formula object.

```
\langle \text{ Extended subexpression implementation } 1390 \rangle + \equiv \\ \textbf{procedure } extSubexpObj.FinishMultiPredicativeFormula;} \\ \textbf{begin } gLastFormula \leftarrow new(MultiPredicativeFormulaPtr, Init(nSubexpPos, new(PList, MoveList(nMultiPredicateList))));} \\ \textbf{end;} \\ \end{cases}
```

1466. The Parser has just parsed " $\langle Term \rangle$ is $\langle Type \rangle$ ", and now we need to store the accumulated data into a Formula AST. Of course, if the *gLastType* variable is not pointing to a type object, then we should raise an error (clearly something has gone wrong somewhere).

If we have accumulated attributes while parsing, then we should update the gLastType to be a clustered type object (and we should move the attributes over).

We should allocate a *QualifiedFormula* object, update the *gLastFormula* state variable to point to it. If the Parser has encountered " $\langle Term \rangle$ is not $\langle Type \rangle$ ", then it will tell the caller to toggle the *nPostNegated* to be true — and in that case, we should negate the *qLastFormula* state variable.

We mutate the TermNbr state variable, decrementing it by one (since we consumed the top of the term stack).

```
\langle \text{ Extended subexpression implementation } 1390 \rangle + \equiv \\ \textbf{procedure } extSubexpObj.FinishQualifyingFormula;} \\ \textbf{var } j: integer; \\ \textbf{begin } mizassert(5430, gLastType \neq \textbf{nil}); \\ \textbf{if } nAttrCollection.Count > 0 \textbf{ then} \\ \textbf{begin } gLastType \leftarrow new(ClusteredTypePtr, Init(gLastType\uparrow.nTypePos, new(PList, Init(nAttrCollection.Count)), gLastType)); \\ \textbf{for } j \leftarrow 0 \textbf{ to } nAttrCollection.Count - 1 \textbf{ do} \\ ClusteredTypePtr(gLastType)\uparrow.nAdjectiveCluster\uparrow.Insert(PObject(nAttrCollection.Items\uparrow[j])); \\ \textbf{end}; \\ gLastFormula \leftarrow new(QualifyingFormulaPtr, Init(nSubexpPos, Term[TermNbr], gLastType)); \\ \textbf{if } nPostNegated \textbf{ then } gLastFormula \leftarrow new(NegativeFormulaPtr, Init(nNotPos, gLastFormula)); \\ dec(TermNbr); \\ \textbf{end}; \\ \end{cases}
```

1467. The Parser has just finished parsing " $\langle Term \rangle$ is $\langle Attribute \rangle$ " or " $\langle Term \rangle$ is not $\langle Attribute \rangle$ ", and so it invokes this method. We allocate a new AttributiveFormula object, and negate it if needed. We also decrement the TermNbr state variable (since we consumed one element of the term stack).

```
 \langle \text{ Extended subexpression implementation } 1390 \rangle + \equiv \\ \textbf{procedure } extSubexpObj.FinishAttributiveFormula; \\ \textbf{begin } gLastFormula \leftarrow new(AttributiveFormulaPtr,Init(nSubExpPos,Term[TermNbr],new(PList,MoveList(nAttrCollection)))); \\ \textbf{if } nPostNegated \textbf{ then } gLastFormula \leftarrow new(NegativeFormulaPtr,Init(nNotPos,gLastFormula)); \\ dec(TermNbr); \\ \textbf{end;} \end{aligned}
```

490 Parsing formulas Mizar Parser $\S1468$

1468. While the Parser is working its way through a formula, and it is looking at an identifier and the next token is a square bracket "[", then the Parser invokes this method to initialize the relevant fields to store accumulated data.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.StartPrivateFormula; 
begin <math>nTermBase \leftarrow TermNbr; nSubexpPos \leftarrow CurPos; nSpelling \leftarrow CurWord.Nr; end;
```

1469. The Parser has just encountered "]" and now we assemble the accumulated data into a formula. This allocates a new *PrivatePredicativeFormula*, moves the arguments encountered since starting the private predicate into a list (§1393) owned by the formula object. The *gLastFormula* is updated to point to the newly allocated formula object.

```
\langle Extended subexpression implementation 1390\rangle += procedure extSubexpObj.FinishPrivateFormula; begin gLastFormula \leftarrow new(PrivatePredicativeFormulaPtr, Init(nSubexpPos, nSpelling, CreateArgs(nTermBase + 1))); end;
```

1470. The Parser has encountered the "contradiction" token, so it invokes this method, which allocates a *ContradictionFormula* and updates the *gLastFormula* state variable to point to it.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.ProcessContradiction; 
begin <math>gLastFormula \leftarrow new(ContradictionFormulaPtr, Init(CurPos)); end;
```

1471. The Parser routinely allocates a formula object, then realizes later it should negate that formula object. This is handled by storing the formula object in the gLastFormula object, then this method allocates a new formula (which is the negation of the gLastFormula) and updates the gLastFormula to point to the newly allocated negated formula.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.ProcessNegative; begin gLastFormula \leftarrow new(NegativeFormulaPtr, Init(CurPos, gLastFormula)); end;
```

1472. When the Parser has encountered the "not" reserved keyword, it invokes the ProcessNegation method which just toggles the nPostNegated field of the caller, and assigns the nNotPos field to the Parser's current position.

```
\langle Extended subexpression implementation 1390 \rangle +\equiv procedure extSubexpObj.ProcessNegation; begin nPostNegated \leftarrow \neg nPostNegated; nNotPos \leftarrow CurPos; end;
```

1473. When the Parser is looking at a binary connective token (e.g., "implies", "or", etc.), this method is invoked to store the connective kind as well as the "left-hand side" to the binary connective in the nFirstSententialOperand field.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.ProcessBinaryConnective; 
begin <math>nConnective \leftarrow CurWord.Kind; nFirstSententialOperand \leftarrow gLastFormula; nSubexpPos \leftarrow CurPos; 
end;
```

 $\S1474$ Mizar Parser PARSING FORMULAS 491

```
The Parser has seen "\(\formula\) or \(\ldots\) or". Then this method will be invoked to store that first
formula parsed in the caller's nFirstSententialOperand field.
\langle Extended subexpression implementation 1390\rangle + \equiv
procedure extSubexpObj.ProcessFlexDisjunction;
  begin nFirstSententialOperand \leftarrow gLastFormula;
  end;
1475.
        The Parser has seen "(Formula) & ... &". Then this method will be invoked to store that first
formula parsed in the caller's nFirstSententialOperand field.
\langle Extended subexpression implementation 1390\rangle + \equiv
procedure extSubexpObj.ProcessFlexConjunction;
  begin nFirstSententialOperand \leftarrow qLastFormula;
  end;
1476.
        The Parser has parsed "for \( \text{Qualified-Variables} \) st", and it is staring at the "st" token. Then it
will invoke this method to mark the nRestrPos, setting it equal to the Parser's current position.
\langle Extended subexpression implementation 1390\rangle + \equiv
procedure extSubexpObj.StartRestriction;
  begin nRestrPos \leftarrow CurPos;
  end;
1477. The Parser has just parsed the formula appearing after "st", so this method is invoked to store
that formula in the caller's nRestriction field (for later use when constructing an AST).
\langle Extended subexpression implementation 1390\rangle + \equiv
procedure extSubexpObj.FinishRestriction;
  begin nRestriction \leftarrow gLastFormula;
  end;
        The Parser has finished parsing a formula involving binary connectives, then it invokes this method
to construct the formula AST.
  If somehow the connective is not "implies", "iff", "or", or "&", then we should raise an error.
\langle Extended subexpression implementation 1390\rangle + \equiv
procedure extSubexpObj.FinishBinaryFormula;
  begin case nConnective of
  sy\_Implies: gLastFormula \leftarrow new(ConditionalFormulaPtr, Init(nSubExpPos, nFirstSententialOperand,
         gLastFormula));
  sy\_Iff: qLastFormula \leftarrow new(BiconditionalFormulaPtr, Init(nSubexpPos, nFirstSententialOperand,
         qLastFormula));
  sy\_Or: gLastFormula \leftarrow new(DisjunctiveFormulaPtr, Init(nSubexpPos, nFirstSententialOperand,
         gLastFormula));
  sy\_Ampersand: gLastFormula \leftarrow new(ConjunctiveFormulaPtr, Init(nSubexpPos,
         nFirstSententialOperand, gLastFormula));
  othercases RunTimeError(3124);
  endcases;
  end:
```

492 Parsing formulas Mizar Parser $\S1479$

1479. We have parsed " $\langle Formula \rangle$ or ... or $\langle Formula \rangle$ ", and the Parser invokes this method to construct an AST for the formula. This method allocates a new FlexaryDisjunctive formula object, and updates the gLastFormula state variable to point to it.

```
the gLastFormula state variable to point to it.

There is a comment in Polish, "polaczyc z flexConj", which Google translates to "connect to flexConj".

⟨Extended subexpression implementation 1390⟩ +≡

procedure extSubexpObj.FinishFlexDisjunction; { polaczyc z flexConj }

begin gLastFormula ← new(FlexaryDisjunctiveFormulaPtr, Init(CurPos, nFirstSententialOperand, gLastFormula));
```

1480. We have parsed " $\langle Formula \rangle$ & ... & $\langle Formula \rangle$ ", and the Parser invokes this method to construct an AST for the formula. This allocates a new FlexaryConjunctive formula object, and updates the gLastFormula state variable to point to it.

```
\label{eq:continuous} \begin{array}{l} \langle \, \text{Extended subexpression implementation 1390} \, \rangle \, + \equiv \\ \textbf{procedure } \, extSubexpObj.FinishFlexConjunction; \\ \textbf{begin } \, gLastFormula \leftarrow new(FlexaryConjunctiveFormulaPtr,Init(CurPos,nFirstSententialOperand, gLastFormula)); \\ \textbf{end;} \end{array}
```

1481. The Parser is looking at the "ex" token, then invokes this method to reset the caller's fields in preparation for accumulating data needed when constructing the formula's AST.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.StartExistential; begin nQualifiedSegments.Init(0); nSubexpPos \leftarrow CurPos; end;
```

end;

1482. The Parser is looking at the "for" token, and it invokes this method to reset the relevant fields in the caller.

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.StartUniversal; begin nQualifiedSegments.Init(0); nSubexpPos \leftarrow CurPos; end;
```

1483. After the Parser has invoked *StartUniversal* or *StartExistential*, it parses the quantified variables (which begins by invoking this method).

```
\langle Extended subexpression implementation 1390\rangle +\equiv procedure extSubexpObj.StartQualifiedSegment; begin nSegmentIdentColl.Init(2); nSegmentPos \leftarrow CurPos; end;
```

1484. The Parser has parsed a comma-separated list and is expecting either "be" or "being", but before parsing for that copula the Parser invokes the *StartQualifyingType* method to update the *gLastType* state variable to point to nil.

```
⟨Extended subexpression implementation 1390⟩ +≡ procedure extSubexpObj.StartQualifyingType; begin gLastType ← nil; end;
```

 $\{1485$ Mizar Parser PARSING FORMULAS 493

1485. The Parser has just finished parsing quantified variables. There are two possible situations:

- (1) We have just parsed reserved variables, so the types are all known. Then the gLastType = nil.
- (2) We have parsed an explicitly typed list of variables, so the gLastType \neq nil.

In the first case, we should allocate an *ImplicitlyQualifiedSegment* object and move all the segment's identifiers to this object. Then we clean up the caller's *nSegmentIdentColl* field (since it's an array of **nil** pointers).

In the second case, we can just move the identifiers when allocating a new Explicitly Qualified Segment object.

In both cases, the new allocated QuantifiedSegment object is appended to the caller's nQualifiedSegments field

```
\langle \text{ Extended subexpression implementation } 1390 \rangle + \equiv \\ \textbf{procedure } extSubexpObj.FinishQualifiedSegment;} \\ \textbf{var } k: integer; \\ \textbf{begin if } gLastType = \textbf{nil then} \\ \textbf{begin for } k \leftarrow 0 \textbf{ to } nSegmentIdentColl.Count - 1 \textbf{ do} \\ \textbf{begin } nQualifiedSegments.Insert(new(ImplicitlyQualifiedSegmentPtr, \\ Init(VariablePtr(nSegmentIdentColl.Items\uparrow[k])\uparrow.nVarPos, nSegmentIdentColl.Items\uparrow[k]))); \\ nSegmentIdentColl.Items\uparrow[k] \leftarrow \textbf{nil}; \\ \textbf{end}; \\ nSegmentIdentColl.Done; \\ \textbf{end} \\ \textbf{else begin } nQualifiedSegments.Insert(new(ExplicitlyQualifiedSegmentPtr, Init(nSegmentPos, \\ new(PList, MoveList(nSegmentIdentColl)), gLastType))); \\ \textbf{end}; \\ \textbf{end
```

1486. When the Parser is parsing quantified variables, specifically when it is parsing a comma-separated list of variables, it will invoke this method, then check if the next token is a comma (and if so iterate). This *Process Variable* method should accumulate a *Variable* object with the current token's identifier, then insert it into the caller's *nSegmentIdentColl* field.

```
⟨ Extended subexpression implementation 1390⟩ +≡
procedure extSubexpObj.ProcessVariable;
begin nSegmentIdentColl.Insert(new(VariablePtr, Init(CurPos, GetIdentifier)));
end;
```

494 Parsing formulas Mizar Parser $\S1487$

```
1487. The Parser has just finished something like
```

```
ex \langle \mathit{Qualified}\text{-}\mathit{Variables} \rangle , \ldots , \langle \mathit{Qualified}\text{-}\mathit{Variables} \rangle st \langle \mathit{Formula} \rangle
```

Now we assemble it as

```
ex \(\langle Qualified\text{-Variables}\rangle\) st \(\((\ext{ex}\)\) \(\text{ormula}\)\)
```

starting with the innermost existentially quantified formula, working our ways outwards. Importantly, assembling the AST reflects the quantified variables has the grammar

```
\langle Qualified\text{-}Variables \rangle = \langle Implicitly\text{-}Qualified\text{-}Variables \rangle
|\langle Explicitly\text{-}Qualified\text{-}Variables \rangle
|\langle Explicitly\text{-}Qualified\text{-}Variables \rangle \text{ "," } \langle Implicitly\text{-}Qualified\text{-}Variables \rangle
```

 $\langle\, \text{Extended subexpression implementation } 1390\,\rangle\,+\!\equiv$

 ${\bf procedure}\ {\it extSubexpObj.FinishExistential};$

```
var k: integer;
```

```
\textbf{begin for } k \leftarrow nQualifiedSegments.Count-1 \ \textbf{downto} \ 1 \ \textbf{do} \quad \{ \text{from inside outwards} \}
```

begin $gLastFormula \leftarrow new(ExistentialFormulaPtr, Init(QualifiedSeqmentPtr(nQualifiedSeqments.Items \uparrow [k]) \uparrow .nSeqmPos,$

 $nQualifiedSegments.Items \uparrow [k], gLastFormula)); nQualifiedSegments.Items \uparrow [k] \leftarrow \mathbf{nil};$

end;

if nQualifiedSegments.Count > 0 then

```
begin gLastFormula \leftarrow new(ExistentialFormulaPtr, Init(nSubexpPos, nQualifiedSegments.Items \uparrow [0], gLastFormula)); nQualifiedSegments.Items <math>\uparrow [0] \leftarrow \mathbf{nil};
```

end:

nQualified Segments. Done;

end:

1488. Universally quantified formulas first transforms

```
for \langle Qualified\text{-}Variables \rangle st \langle Formula \rangle_1 holds \langle Formula \rangle_2
```

into

for
$$\langle Qualified\text{-}Variables \rangle$$
 holds $\langle Formula \rangle_1$ implies $\langle Formula \rangle_2$

which is handled immediately.

The remainder of the method iteratively constructs the universally quantified formulas by "unrolling" the qualified segments, just as we did for existentially quantified formulas.

```
⟨ Extended subexpression implementation 1390⟩ +≡
```

 ${\bf procedure}\ extSubexpObj.FinishUniversal;$

```
var k: integer;
```

```
begin if nRestriction \neq nil then {transform st into implies}
```

 $gLastFormula \leftarrow new(ConditionalFormulaPtr, Init(nRestrPos, nRestriction, gLastFormula)); \\$

 $\textbf{for } k \leftarrow nQualifiedSegments.Count-1 \ \textbf{downto} \ 1 \ \textbf{do}$

 $\mathbf{begin} \ gLastFormula \leftarrow new(\mathit{UniversalFormulaPtr},$

 $Init(Qualified Segment Ptr(nQualified Segments. Items \uparrow [k]) \uparrow. nSegmPos,\\$

 $nQualifiedSegments.Items \uparrow [k], gLastFormula)); \ nQualifiedSegments.Items \uparrow [k] \leftarrow \mathbf{nil};$ end:

if nQualifiedSegments.Count > 0 then

```
 \begin{array}{l} \textbf{begin} \ gLastFormula \leftarrow new (\textit{UniversalFormulaPtr}, \textit{Init} (nSubexpPos, nQualifiedSegments.Items \uparrow [0], \\ gLastFormula)); \ nQualifiedSegments.Items \uparrow [0] \leftarrow \textbf{nil}; \end{array}
```

end; end;

Section 23.4. EXTENDED EXPRESSION CLASS

begin $gSubexpPtr \leftarrow new(extSubexpPtr, Init)$

end;

1489. When an expression is needed, the gExpPtr state variable is used to build it out of subexpressions. The *qExpPtr* state variable is an instance of the *extExpression* class. \langle Extended expression class declaration 1489 $\rangle \equiv$ $extExpressionPtr = \uparrow extExpressionObj;$ extExpressionObj = object (ExpressionObj)**constructor** *Init*(*fExpKind* : *ExpKind*); procedure CreateSubexpression; virtual; end; This code is used in section 1199. 1490. Constructor. This just invokes the parent class's constructor (§711), then resets the module-wide variable TermNbr to zero. \langle Extended expression implementation $1490 \rangle \equiv$ **constructor** *extExpressionObj.Init(fExpKind : ExpKind)*; **begin** $inherited Init(fExpKind); TermNbr \leftarrow 0;$ end; See also section 1491. This code is used in section 1200. An extExpression creating a subexpression overrides the parent class's method ($\S712$), and sets the global gSubexpPtr to point to a new extSubexp object. \langle Extended expression implementation $1490 \rangle + \equiv$ **procedure** *extExpressionObj.CreateSubexpression*;

496 PARSER Mizar Parser $\S1492$

File 24

Parser

1492. The parser has a "big red button": a single "obvious" function for the user to, you know, push. Namely, the *Parse* procedure ($\S\S1658$ et seq.). Everything else is just a helper function.

The design of the parser appears to be a recursive descent parser on statements, with parsing expressions handled specially.

Note that the base/parser.pas file appears to be naturally divided up into sections, with comments which appear to use the Germanic "s p a c i n g f o r i t a l i c s" (which I have just replaced with more readable *italicized* versions). I have used these cleavages to organize the discussion of this file.

The *StillCorrect* global variable is *false* when the parser has entered what programmers call "Panic Mode": something has gone awry, and the parser is trying to recover gracefully. For a friendly review of panicking, see Bob Nystrom's *Crafting Interpreters* (Chaper 6, Section 3).

```
⟨ parser.pas 1492⟩ ≡
⟨ GNU License 4⟩
unit parser;
interface
uses mscanner;
var StillCorrect: boolean = true;
type ReadTokenProcedure = Procedure;
const ReadTokenProce: ReadTokenProcedure = ReadToken; {from mscanner.pas}
procedure Parse;
procedure SemErr(fErrNr : integer);
implementation
uses syntax, errhan, pragmas
    mdebug , info end_mdebug;
⟨ Implementation of parser.pas 1493⟩
```

1493. We have a few constants, but the implementation is loosely organized around parsing expressions (terms and formulas), statements, and then blocks.

```
⟨ Implementation of parser.pas 1493⟩ ≡
  ⟨ Local constants for parser.pas 1494⟩;
  ⟨ Parse expressions (parser.pas) 1502⟩
  ⟨ Communicate with items (parser.pas) 1558⟩
  ⟨ Process miscellany (parser.pas) 1559⟩
  ⟨ Parse simple justifications (parser.pas) 1571⟩
  ⟨ Parse statements and reasoning (parser.pas) 1577⟩
  ⟨ Parse patterns (parser.pas) 1602⟩
  ⟨ Parse definitions (parser.pas) 1621⟩
  ⟨ Parse scheme block (parser.pas) 1654⟩
  ⟨ Main parse method (parser.pas) 1658⟩
See also sections 1495, 1496, 1498, 1499, 1500, and 1501.
This code is used in section 1492.
```

§1494 PARSER 497 Mizar Parser

1494. We have error codes for syntactically invalid situations. These are all different ways for panic to occur (hence the "pa-" prefix).

```
\langle Local constants for parser.pas 1494 \rangle \equiv
const paUnexpOf = 183; paUnexpOver = 184; paUnexpEquals = 186; paUnexpAntonym1 = 198;
  paUnexpAntonym2 = 198; paUnexpSynonym = 199; paUnpairedSymbol = 214; paEndExp = 215;
  paUnexpHereby = 216; paAdjClusterExp = 223; paUnexpReconsider = 228; paPerExp = 231;
  paSupposeOrCaseExp = 232; paOfExp = 256; paUnexpRedef = 273; paAllExp = 275;
  paIdentExp1 = 300; paIdentExp2 = 300; paIdentExp3 = 300; paIdentExp4 = 300; paIdentExp5 = 300;
  paIdentExp6 = 300; paIdentExp7 = 300; paIdentExp8 = 300; paIdentExp9 = 300; paIdentExp10 = 300;
  paIdentExp11 = 300; paIdentExp12 = 300; paIdentExp13 = 300; paWronqPredPattern = 301;
  paFunctExp1 = 302; paFunctExp2 = 302; paFunctExp3 = 302; paFunctExp4 = 302;
  paWrongModePatternBeg = 303; paStructExp1 = 304; paSelectExp1 = 305; paAttrExp1 = 306;
  paAttrExp2 = 306; paAttrExp3 = 306; paNumExp = 307; paWrongReferenceBeq = 308;
  paTypeOrAttrExp = 309; paRightBraExp1 = 310; paRightBraExp2 = 310;
  paWrongRightBracket1 = 311; paWrongRightBracket2 = 311; paDefExp = 312; paSchExp = 313;
  paWronqPattBeq1 = 314; paWronqPattBeq2 = 314; paWronqPattBeq3 = 314;
  paWrongModePatternSet=315;\ paWrongAfterThe=320;\ paWrongPredSymbol=321;
  paSemicolonExp = 330; \ paUnexpConnective = 336; \ paWrongScopeBeg = 340; \ paThatExp1 = 350;
  paThatExp2 = 350; paCasesExp = 351; paLeftParenthExp = 360; paLeftSquareExp = 361;
  paLeftCurledExp = 362; paLeftDoubleExp1 = 363; paLeftDoubleExp3 = 363;
  paWrongSchemeVarQual = 364; paRightParenthExp1 = 370; paRightParenthExp2 = 370;
  paRightParenthExp3 = 370; paRightParenthExp4 = 370; paRightParenthExp5 = 370;
  paRightParenthExp6 = 370; paRightParenthExp7 = 370; {forgot right paren in "from SCHEME("}
  paRightParenthExp8 = 370; paRightParenthExp9 = 370; paRightParenthExp10 = 370;
  paRightParenthExp11 = 370; paRightSquareExp1 = 371; paRightSquareExp2 = 371;
  paRightSquareExp3 = 371; paRightSquareExp4 = 371; paRightSquareExp5 = 371;
  paRightCurledExp1 = 372; paRightCurledExp2 = 372; paRightCurledExp3 = 372;
  paRightDoubleExp1 = 373; paRightDoubleExp2 = 373; paWrongAttrPrefixExpr = 375;
  paWrongAttrArqumentSuffix = 376; paTypeExpInAdjectiveCluster = 377; paEqualityExp1 = 380;
  paEqualityExp2 = 380; paIfExp = 381; paForExp = 382; paIsExp = 383; paColonExp1 = 384;
  paColonExp2 = 384; paColonExp3 = 384; paColonExp4 = 384; paArrowExp1 = 385;
  paArrowExp2 = 385; paMeansExp = 386; paStExp = 387; paAsExp = 388; paProofExp = 389;
  paWithExp = 390; paWrongItemBeq = 391; paUnexpItemBeq = 392; paWrongJustificationBeq = 395;
  paWrongFormulaBeg = 396; paWrongTermBeg = 397; paWrongRadTypeBeg = 398;
  paWronqFunctorPatternBeq = 399; paStillNotImplemented = 400; paNotExpected = 401;
  paInfinitiveExp = 402; \ paSuchExp = 403; \ paToExp = 404; \ paTypeUnexpInClusterRegistration = 405;
  paForOrArrowExpected = 406;
```

See also section 1497.

This code is used in section 1493.

```
1495. (Implementation of parser.pas 1493) +\equiv
var gAddSymbolsSet: set of char = []; { not used anywhere }
```

498 PARSER Mizar Parser §1496

```
1496.
        Syntax errors do three things:
(1) Marks StillCorrect to be false (i.e., enters panic mode)
(2) Reports the error with the ErrImm (§106) function.
(3) Skips ahead until we find a token in the gMainSet, then try to proceed like things are still alright (so
    we "fail gracefully").
\langle Implementation of parser.pas 1493\rangle + \equiv
procedure SynErr(fPos: Position; fErrNr: integer);
  begin if StillCorrect then
    begin StillCorrect \leftarrow false;
    if CurWord.Kind = sy\_Error then
       begin if CurWord.Nr \neq scTooLongLineErrorNr then ErrImm(CurWord.Nr)
       else Error(fPos, fErrNr);
       end
    else Error(fPos, fErrNr);
    while \neg (CurWord.Kind \in gMainSet) do ReadTokenProc;
    end;
  end;
        What constants are good "check-in points" for the parser to recover at? The beginning of blocks,
the end of statements (especially semicolons), and the end of text.
\langle \text{Local constants for parser.pas } 1494 \rangle + \equiv
const gMainSet: set of TokenKind = [sy\_Begin, sy\_Semicolon, sy\_Proof, sy\_Now, sy\_Hereby,
         sy\_Definition, sy\_End, sy\_Theorem, sy\_Reserve, sy\_Notation, sy\_Registration, sy\_Scheme, EOT,
         sy\_Deffunc, sy\_Defpred, sy\_Reconsider, sy\_Consider, sy\_Then, sy\_Per, sy\_Case, sy\_Suppose];
1498. We have a few more methods for specific kinds of errors we are likely to encounter.
\langle \text{Implementation of parser.pas } 1493 \rangle + \equiv
procedure Missing Word (fErrNr: integer);
  var lPos: Position;
  begin lPos \leftarrow PrevPos; inc(lPos.Col); SynErr(lPos,fErrNr)
  end:
procedure Wrong Word (fErrNr: integer);
  begin SynErr(CurPos, fErrNr)
  end:
1499. We will want to assert the parser has encountered a specific token (like a semicolon or "end") and
raise an error if it has not. This will make for much more readable code later on. We should recall KillItem
(\S686) mutates the global state.
\langle Implementation of parser.pas 1493 \rangle + \equiv
procedure Semicolon;
  begin KillItem:
  if CurWord.Kind \neq sy\_Semicolon then MissingWord(paSemicolonExp);
  if CurWord.Kind = sy\_Semicolon then ReadTokenProc;
  end;
procedure AcceptEnd(fPos:Position);
  begin if CurWord.Kind = sy\_End then ReadTokenProc
  else begin Error (fPos, paEndExp); MissingWord (paUnpairedSymbol)
  end;
```

 $\S1500$ Mizar Parser PARSER 499

1500. These previous methods can be generalized to an Accept procedure which checks whether a given TokenKind has "occurred". If so, just read the next word. Otherwise, flag an error. $\langle Implementation of parser.pas 1493 \rangle +\equiv$ **procedure** ReadWord;

```
procedure ReadWord;

begin Mizassert(2546, StillCorrect); ReadTokenProc

end;

function Occurs(fW: TokenKind): boolean;

begin Occurs \leftarrow false;

if CurWord.Kind = fW then

begin ReadWord; Occurs \leftarrow true

end

end;

procedure Accept(fCh: TokenKind; fErrNr: integer);

begin if \neg Occurs(fCh) then MissingWord(fErrNr)

end;
```

1501. Flagging a semantic error should first check if we are in "panic mode" or not. If we are already panicking, there's no reason to heap more panicky error messages onto the screen.

```
⟨Implementation of parser.pas 1493⟩ +≡
procedure SemErr(fErrNr : integer);
begin if StillCorrect then ErrImm(fErrNr)
end;
```

500 EXPRESSIONS Mizar Parser $\S1502$

Section 24.1. EXPRESSIONS

```
We have a few token kinds which indicate the start of a term:
(1) identifiers (for variables and private functors),
(2) infixed operators,
(3) numerals,
(4) left and right brackets of all sorts,
(5) the anaphoric "it" constant used in definitions,
(6) "the" choice operator,
(7) placeholder variables appearing in private functors and predicates,
(8) structure symbols.
\langle \text{Parse expressions (parser.pas) } 1502 \rangle \equiv
     \{Expressions\}
const TermBegSys: set of
     TokenKind = [Identifier, InfixOperatorSymbol, Numeral, LeftCircumfixSymbol, sy_LeftParanthesis, ]
          sy\_It, sy\_LeftCurlyBracket, sy\_LeftSquareBracket, sy\_The, sy\_Dolar, Structuresymbol\};
See also sections 1503, 1504, 1505, 1506, 1508, 1518, 1519, 1520, 1523, 1525, 1530, 1531, 1533, 1534, 1535, 1537, 1538, 1539,
     1540, 1543, 1544, 1549, 1552, 1553, 1554, 1555, 1556, and 1557.
This code is used in section 1493.
```

Subsection 24.1.1. Terms

1503. We have a few helper function for Accept-ing parentheses. This invokes the ProcessLeftParenthesis method for the gSubexpPtr (§690) global variable which we recall (§718) is an empty virtual method. So the parser just "consumes" a left parentheses, and will continue to read tokens while they are left parentheses. The argument passed in will be mutated to track the number of left parentheses consumed.

Similarly, the *CloseParenth* method will have the compiler consume right parentheses, mutating the argument passed in (to decrement the number of right parentheses consumed). This will let us track mismatched parentheses errors.

```
⟨ Parse expressions (parser.pas) 1502 ⟩ +≡
procedure OpenParenth(var fParenthCnt : integer);
begin fParenthCnt ← 0;
while CurWord.Kind = sy_LeftParanthesis do
   begin gSubexpPtr↑.ProcessLeftParenthesis; ReadWord; inc(fParenthCnt);
   end;
end;
procedure CloseParenth(var fParenthCnt : integer);
begin while (CurWord.Kind = sy_RightParanthesis) ∧ (fParenthCnt > 0) do
   begin dec(fParenthCnt); gSubexpPtr↑.ProcessRightParenthesis; ReadWord;
   end;
end;
```

 $\{1504$ Mizar Parser TERMS 501

1504. Qualified expressions. Parsing qualified expressions includes a control flow for "exactly" qualified expressions.

We should recall from "Mizar in a nutshell" that the "exactly" keyword is reserved but not currently used for anything. The global subexpression pointer is invoking empty virtual methods (§718). So what's going on?

Well, the only work being done here is in the branch handling "qua", specifically the next word is read, and then control is handed off to TypeSubexpression.

```
⟨ Parse expressions (parser.pas) 1502 ⟩ +≡
procedure TypeSubexpression; forward;

procedure AppendQua;
begin while CurWord.Kind = sy_Qua do
   begin gSubexpPtr↑.ProcessQua; ReadWord; TypeSubexpression; gSubexpPtr↑.FinishQualifiedTerm;
   end;
if CurWord.Kind = sy_Exactly then
   begin gSubexpPtr↑.ProcessExactly; ReadWord
   end;
end;
```

1505. Parsing the contents of a bracketed term starts a bracketed term ($\S718$), reads the next word after the start of the bracket, then consumes the maximum number of visible arguments ($\S682$).

The contract for this function is that a left bracket token has been encountered, the parser has moved on to the next token, and then invoked this function.

```
\langle \text{Parse expressions (parser.pas) } 1502 \rangle + \equiv procedure GetArguments(\mathbf{const}\ fArgsNbr:\ integer);\ forward; procedure BracketedTerm; begin gSubexpPtr\uparrow.StartBracketedTerm;\ ReadWord;\ GetArguments(MaxVisArgNbr);\ gSubexpPtr\uparrow.FinishBracketedTerm; end;
```

1506. Parsing post-qualified variables (i.e., variables which appear in a Fraenkel term's "where" clause) which consists of a comma-separated list of post-qualified segments.

```
⟨ Parse expressions (parser.pas) 1502⟩ +≡
procedure TermSubexpression; forward;
procedure FormulaSubexpression; forward;
procedure ArgumentsTail(fArgsNbr:integer); forward;
procedure ProcessPostqualification;
begin gSubexpPtr↑.StartPostqualification;
while CurWord.Kind = sy_Where do
   begin repeat ⟨ Process post-qualified segment 1507⟩
   until CurWord.Kind ≠ sy_Comma;
   end;
end;
```

502 TERMS Mizar Parser $\S1507$

1507. Each "segment" in a post-qualification looks like:

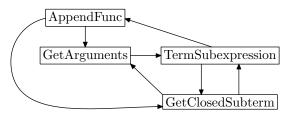
This code is used in section 1506.

```
\langle variable \rangle  {"," \langle variable \rangle \} ("is" | "being") \langle type \rangle
```

We can process the comma-separated list of variables, then the type ascription term ("is" or "being"), then process the type.

```
 \begin{array}{l} \textbf{define} \ \ process\_postqualified\_variables \equiv \textbf{repeat} \ \ gSubexpPtr \uparrow. ProcessPostqualifiedVariable; \\ Accept (Identifier, paIdentExp1); \\ \textbf{until} \ \neg Occurs (sy\_Comma) \\ \textbf{define} \ \ post\_qualified\_type \equiv \textbf{begin} \ \ ReadWord; \ \ TypeSubexpression; \textbf{end} \\ \langle \operatorname{Process} \ post\_qualified \ segment \ \ 1507 \rangle \equiv \\ gSubexpPtr \uparrow. StartPostQualifyingSegment; \ \ ReadWord; \\ process\_postqualified\_variables; \\ gSubexpPtr \uparrow. StartPostqualificationSpecyfication; \\ \textbf{if} \ \ CurWord. Kind \in [sy\_Is, sy\_are] \ \textbf{then} \ \ post\_qualified\_type; \\ gSubexpPtr \uparrow. FinishPostqualifyingSegment; \end{aligned}
```

1508. Getting a closed subterm is part of the loop for parsing a term. The intricate relationship of mutually recursive function calls looks something like the following (assuming there are no parsing errors):



 ${\bf Fig.~8.}$ Control flow when parsing a term.

The GetArguments parses a comma-separated list of terms. Since each term in the comma-separated list will be a subterm of a larger expression, we parse it with TermSubexpression (which invokes GetClosedSubterm in a mutually recursive relation). If there is a chain of infix operators (like $x + y - z \times \omega$), then AppendFunc is invoked on the infixed operators.

```
⟨ Parse expressions (parser.pas) 1502⟩ +≡
procedure GetClosedSubterm;
begin case CurWord.Kind of
  ⟨ Get closed subterm of identifier 1509⟩;
  ⟨ Get closed subterm of structure 1510⟩;
Numeral: begin gSubexpPtr↑.ProcessNumeralTerm; ReadWord end;
  ⟨ Get closed subterm of bracketed expression 1511⟩;
sy_It: begin gSubexpPtr↑.ProcessItTerm; ReadWord end;
sy_Dolar: begin gSubexpPtr↑.ProcessLocusTerm; ReadWord end;
  ⟨ Get closed subterm of Fraenkel operator or enumerated set 1512⟩;
  ⟨ Get closed subterm of choice operator 1515⟩;
othercases RunTimeError(2133);
endcases;
end;
```

 $\{1509$ Mizar Parser TERMS 503

```
1509. If we treat an identifier as a term, then it is either a private functor or it is a variable. How do we tell the difference? A private functor starts with an identifier followed by a left parentheses.
⟨Get closed subterm of identifier 1509⟩ ≡
Identifier: if AheadWord.Kind = sy_LeftParanthesis then { treat identifier as private functor } begin gSubexpPtr↑.StartPrivateTerm; ReadWord; ReadWord; if CurWord.Kind ≠ sy_RightParanthesis then GetArguments(MaxVisArgNbr); gSubexpPtr↑.FinishPrivateTerm; Accept(sy_RightParanthesis, paRightParenthExp2); end else { treat identifier as variable } begin gSubexpPtr↑.ProcessSimpleTerm; ReadWord end
This code is used in section 1508.
1510. If the parser stumbles across the name of a structure when expecting a term, then the parser should treat it as constructing a new instance of the structure.
```

```
\langle \mbox{ Get closed subterm of structure } 1510 \rangle \equiv StructureSymbol: \mbox{ begin } gSubexpPtr \uparrow. StartAggregateTerm; \ ReadWord; \\ Accept(sy\_StructLeftBracket, paLeftDoubleExp1); \ GetArguments(MaxVisArgNbr); \\ gSubexpPtr \uparrow. FinishAggregateTerm; \ Accept(sy\_StructRightBracket, paRightDoubleExp1); \\ \mbox{ end } \mbox{ end } \mbox{ } \mbox{
```

This code is used in section 1508.

1511. Encountering a left bracket of some kind should cause the parser to look for the contents of a bracketed term (§1505), then a right bracket.

```
 \langle \text{ Get closed subterm of bracketed expression 1511} \rangle \equiv \\ LeftCircumfixSymbol, sy\_LeftSquareBracket: \textbf{begin} \ BracketedTerm; \\ \textbf{case} \ Curword.Kind \ \textbf{of} \\ sy\_RightSquareBracket, sy\_RightCurlyBracket, sy\_RightParanthesis: ReadWord; \\ \textbf{othercases} \ Accept(RightCircumfixSymbol, paRightBraExp1); \\ \textbf{endcases}; \\ \textbf{end}
```

This code is used in section 1508.

1512. When the parser runs into a left curly bracket "{", we either have encountered a Fraenkel operator or we have encountered a finite set.

```
\langle Get closed subterm of Fraenkel operator or enumerated set 1512\rangle \equiv sy\_LeftCurlyBracket: begin gSubexpPtr\uparrow.StartBracketedTerm; ReadWord; TermSubexpression; if (CurWord.Kind = sy\_Colon) \lor (CurWord.Kind = sy\_Where) then \langle Parse a Fraenkel operator 1513\rangle else \langle Parse an enumerated set 1514\rangle; end
```

This code is used in section 1508.

1513. Parsing a Fraenkel operator, well, we recall Fraenkel operators look like

```
\{\langle term \rangle \langle post\text{-}qualified \ segment \rangle \ ":" \ \langle formula \rangle \}
```

```
\langle \text{Parse a Fraenkel operator } 1513 \rangle \equiv  begin gSubexpPtr\uparrow.StartFraenkelTerm; ProcessPostqualification; <math>gSubexpPtr\uparrow.FinishSample; Accept(sy\_Colon, paColonExp1); FormulaSubexpression; <math>gSubexpPtr\uparrow.FinishFraenkelTerm; Accept(sy\_RightCurlyBracket, paRightCurledExp1); end
```

This code is used in section 1512.

504 TERMS Mizar Parser $\S1514$

1514. We can also run into a finite set $\{x_1, \ldots, x_n\}$.

```
\langle \text{ Parse an enumerated set 1514} \rangle \equiv
  begin gSubexpPtr\uparrow.FinishArgument; ArgumentsTail(MaxVisArgNbr - 1);
  qSubexpPtr\uparrow.FinishBracketedTerm;
  case Curword.Kind of
  sy_RightSquareBracket, sy_RightCurlyBracket, sy_RightParanthesis: ReadWord;
  othercases Accept(RightCircumfixSymbol, paRightBraExp1);
  endcases;
  end
This code is used in section 1512.
1515. Mizar allows "the" to be used for selector functors, forgetful functors, choice operators, or simple
Fraenkel terms.
\langle Get closed subterm of choice operator 1515\rangle \equiv
sy\_The: \mathbf{begin} \ gSubexpPtr \uparrow. ProcessThe; \ ReadWord;
  case CurWord.Kind of
  SelectorSymbol: begin gSubexpPtr\\capp.StartSelectorTerm; ReadWord;
     if Occurs(sy\_Of) then TermSubexpression;
     gSubexpPtr \uparrow . FinishSelectorTerm;
     end;
  StructureSymbol: (Parse forgetful functor or choice of structure type 1516);
  sy\_Set: \langle Parse simple Fraenkel expression or "the set" 1517 <math>\rangle;
  ModeSymbol, AttributeSymbol, sy_Non, sy_LeftParanthesis, Identifier, InfixOperatorSymbol, Numeral,
          LeftCircumfixSymbol, sy\_It, sy\_LeftCurlyBracket, sy\_LeftSquareBracket, sy\_The, sy\_Dolar:
     begin gSubexpPtr\uparrow.StartChoiceTerm; TypeSubexpression; <math>gSubexpPtr\uparrow.FinishChoiceTerm;
  othercases begin gSubexpPtr\uparrow.InsertIncorrTerm; WrongWord(paWrongAfterThe) end;
  endcases;
  end
This code is used in section 1508.
1516. A forgetful functor always looks like
                                        "the" \langle structure \rangle "of" \langle term \rangle
On the other hand, the choice operator acting on a structure type looks similar. We should distinguish these
two by the presence of the keyword "of".
\langle Parse forgetful functor or choice of structure type 1516\rangle \equiv
  if AheadWord.Kind = sy\_Of then { forgetful functor }
     begin gSubexpPtr\uparrow.StartForgetfulTerm; ReadWord; Accept(sy\_Of, paOfExp); TermSubexpression;
     gSubexpPtr\uparrow.FinishForgetfulTerm;
     end
          { choice operator }
  begin qSubexpPtr\uparrow.StartChoiceTerm; TypeSubexpression; <math>qSubexpPtr\uparrow.FinishChoiceTerm;
  end
This code is used in section 1515.
```

 $\S1517$ Mizar Parser TERMS 505

Mizar allows "the set of" to start a simple Fraenkel expression. But we could also refer to "the

set" as the set chosen by the axiom of choice.
⟨ Parse simple Fraenkel expression or "the set" 1517⟩ ≡
if AheadWord.Kind = sy_Of then { simple Fraenkel expression }
begin ReadWord; { set }
ReadWord; { of }
gSubexpPtr↑.StartSimpleFraenkelTerm; Accept(sy_All, paAllExp); TermSubexpression;
gSubexpPtr↑.StartFraenkelTerm; ProcessPostqualification; gSubexpPtr↑.FinishSimpleFraenkelTerm;
end
else { "the set" }
begin gSubexpPtr↑.StartChoiceTerm; TypeSubexpression; gSubexpPtr↑.FinishChoiceTerm; end
This code is used in section 1515.
1518. Subexpression object's FinishArgument (§1405) is invoked. This will invoke the AppendQua (§1504)

1518. Subexpression object's FinishArgument (§1405) is invoked. This will invoke the AppendQua (§1504 method and expect a closed parentheses afterwards (§1503).

Possible bug: what should happen when fParenthCnt is zero or negative?

```
\langle \text{Parse expressions (parser.pas) } 1502 \rangle + \equiv
procedure CompleteArgument(\mathbf{var}\ fParenthCnt: integer);
begin gSubexpPtr\uparrow.FinishArgument;
repeat AppendQua;\ CloseParenth(fParenthCnt);
until CurWord.Kind \neq sy\_Qua;\ \{\land (CurWord.Kind \neq sy\_Exactly)\}
end;
```

506 TERMS Mizar Parser $\S1519$

1519. Keep parsing "infixed operators". When the current token is an infixed operator, this will consume the arguments to its right, then iterate. It's also worth remembering that gExpPtr (§690) was a global variable declared back in syntax.pas, and the CreateSubexpression (§1491) mutates the gSubexpPtr variable. Now we see it in action.

This invokes the ProcessLeftParenthesis method for the gSubexpPtr (§690) global variable which we recall (§718) is an empty virtual method. So the parser just "consumes" a left parentheses.

Note that the **case** expression considers the type of TokenKind (§724) of the current word. But it is not exhaustive.

There is a comment in Polish, "Chyba po prostu TermSubexpression", which Google translated into English as "I guess it's just Term Subexpression". I swapped this in the code below.

```
\langle \text{Parse expressions (parser.pas) } 1502 \rangle + \equiv
procedure AppendFunc(var fParenthCnt: integer);
  begin while CurWord.Kind = InfixOperatorSymbol do
    begin gSubexpPtr \uparrow . StartLongTerm; { (§1413) }
    repeat qSubexpPtr\uparrow.ProcessFunctorSymbol; { <math>(\S1429) }
       ReadWord:
       case CurWord.Kind of
       sy\_LeftParanthesis:
         begin
                   { parenthetised term(s) }
         gSubexpPtr\uparrow.ProcessLeftParenthesis; ReadWord; {consume the left paren}
         GetArguments(MaxVisArgNbr); \{ (\S1534) \}
         gSubexpPtr\uparrow.ProcessRightParenthesis;\ Accept(sy\_RightParenthesis,paRightParenthExp3);
              { consume matching right paren }
         end:
       Identifier, Numeral, LeftCircumfixSymbol, sy\_It, sy\_LeftCurlyBracket, sy\_LeftSquareBracket, sy\_The,
              sy_Dolar, StructureSymbol: { I guess it's just Term Subexpression }
         begin gExpPtr\uparrow.CreateSubexpression; { (§1491) }
         GetClosedSubterm; \{ (\S1508) \}
         qSubexpPtr\uparrow.FinishArgument; \{ (\S1405) \}
         KillSubexpression; \{ (\S 684) \}
         end:
       endcases;
       gSubexpPtr\uparrow.FinishArgList; \{ (\S1430) \}
    until CurWord.Kind \neq InfixOperatorSymbol;
    gSubexpPtr\uparrow.FinishLongTerm; \{ (\S1417) \}
    CompleteArgument(fParenthCnt); \{ (\S1518) \}
    end;
  end;
```

 $\{1520$ Mizar Parser TERMS 507

```
1520. Parse terms with infix operators. Note this appears to parse infixed operators as left-associative
(e.g., x + y + z is parsed as (x + y) + z).
\langle \text{Parse expressions (parser.pas) } 1502 \rangle + \equiv
procedure ProcessArguments;
  var lParenthCnt: integer;
  begin OpenParenth(lParenthCnt);
  case CurWord.Kind of
  Identifier, Numeral, LeftCircumfixSymbol, sy\_It, sy\_LeftCurlyBracket, sy\_LeftSquareBracket, sy\_The,
         sy\_Dolar, StructureSymbol:
    begin GetClosedSubterm: CompleteArgument(lParenthCnt): end:
  InfixOperatorSymbol:;
  othercases begin qSubexpPtr\uparrow.InsertIncorrTerm; <math>qSubexpPtr\uparrow.FinishArqument;
     WrongWord(paWrongTermBeg);
    end:
  endcases;
  (Keep parsing as long as there is an infixed operator to the right 1521);
  (Check every remaining open (left) parentheses has a corresponding partner 1522);
  end:
1521.
       \langle Keep parsing as long as there is an infixed operator to the right 1521 \rangle \equiv
  repeat AppendFunc(lParenthCnt);
    if CurWord.Kind = sy\_Comma then
       begin ArgumentsTail(MaxVisArgNbr-1);
       if (lParenthCnt > 0) \land (CurWord.Kind = sy\_RightParanthesis) then
         begin dec(lParenthCnt); gSubexpPtr\uparrow.ProcessRightParenthesis; ReadWord;
         end;
       end:
  until CurWord.Kind \neq InfixOperatorSymbol
This code is used in section 1520.
1522. (Check every remaining open (left) parentheses has a corresponding partner 1522) \equiv
  while lParenthCnt > 0 do
    begin qSubexpPtr\uparrow.ProcessRightParenthesis; Accept(<math>sy\_RightParenthesis, paRightParenthExp1);
    dec(lParenthCnt);
    end
This code is used in section 1520.
1523. An adjective cluster is just one or more (possibly negated) attribute.
\langle \text{Parse expressions (parser.pas) } 1502 \rangle + \equiv
  (Process attributes (parser.pas) 1524)
procedure GetAdjectiveCluster;
  begin qSubexpPtr\uparrow.StartAdjectiveCluster; ProcessAttributes; <math>qSubexpPtr\uparrow.FinishAdjectiveCluster;
  end;
```

508 TERMS Mizar Parser $\S1524$

```
1524. Parsing an attribute amounts to:
(1) handling a leading "non"
(2) handling attribute arguments (which always occurs before the attribute)
(3) handling the attribute.
  define kind\_is\_radix\_type(\#) \equiv (\# \in [sy\_Set, ModeSymbol, StructureSymbol])
  define ahead\_is\_attribute\_argument \equiv
         (CurWord.Kind \in (TermBegSys - [sy\_LeftParanthesis, StructureSymbol])) \lor
              ((CurWord.Kind = sy\_LeftParanthesis) \land \neg(kind\_is\_radix\_type(AheadWord.Kind))) \lor
              ((CurWord.Kind = StructureSymbol) \land (AheadWord.Kind = sy\_StructLeftBracket))
\langle Process attributes (parser.pas) 1524 \rangle \equiv
procedure ProcessAttributes;
  begin while (CurWord.Kind \in [AttributeSymbol, sy_Non]) \lor ahead\_is\_attribute\_argument do
    begin gSubexpPtr\uparrow.ProcessNon;
    if CurWord.Kind = sy\_Non then ReadWord;
    {f if}~ahead\_is\_attribute\_argument~{f then}
       begin gSubexpPtr \uparrow. StartAttributeArguments; ProcessArguments;
       gSubexpPtr \uparrow. CompleteAttributeArguments;
       end;
    if CurWord.Kind = AttributeSymbol then
       begin qSubexpPtr\uparrow.ProcessAttribute; ReadWord; end
    else begin SynErr(CurPos, paAttrExp1) end;
    end;
  end;
This code is used in section 1523.
        Parsing a radix type. For Mizar, a Radix type is either a structure type or a mode (or it's the
"set" type).
  There is a comment in Polish, "zawieszone na czas zmiany semantyki", which is translated into English.
\langle \text{Parse expressions (parser.pas)} 1502 \rangle + \equiv
procedure RadixTypeSubexpression;
  var lSymbol, lParenthCnt: integer;
  begin lParenthCnt \leftarrow 0; \langle Parse optional left-paren 1528 \rangle;
  gSubexpPtr\uparrow.ProcessModeSymbol; \{(\S1408)\}
  case CurWord.Kind of
  sy\_Set: begin ReadWord;
         {? if Occurs(syOf) then TypeSubexpression suspended while semantics change}
    end;
  ModeSymbol: \langle Parse mode as radix type 1526 \rangle;
  StructureSymbol: (Parse structure as radix type 1527);
  othercases begin MissingWord(paWrongRadTypeBeg); gSubexpPtr\uparrow.InsertIncorrType end;
  endcases;
  \langle Close the parentheses 1529\rangle;
  qSubexpPtr\uparrow.FinishType;
  end;
```

 $\S1526$ Mizar Parser TERMS 509

```
1526.
         \langle \text{ Parse mode as radix type 1526} \rangle \equiv
  begin lSymbol \leftarrow CurWord.Nr; ReadWord;
  if CurWord.Kind = sy\_Of then
     if ModeMaxArgs.fList \uparrow [lSymbol] = 0 then WrongWord(paUnexpOf)
     else begin ReadWord; GetArguments(ModeMaxArgs.fList \uparrow [lSymbol]) end;
  end
This code is used in section 1525.
1527. \langle Parse structure as radix type 1527 \rangle \equiv
  begin lSymbol \leftarrow CurWord.Nr; ReadWord;
  if CurWord.Kind = sy\_Over then
     if StructModeMaxArgs.fList \uparrow [lSymbol] = 0 then WrongWord(paUnexpOver)
     else begin ReadWord; GetArguments(StructModeMaxArgs.fList \uparrow [lSymbol]) end;
  end
This code is used in section 1525.
1528. \langle \text{Parse optional left-paren 1528} \rangle \equiv
  if CurWord.Kind = sy\_LeftParanthesis then
     begin gSubexpPtr\uparrow.ProcessLeftParenthesis; ReadWord; inc(lParenthCnt);
This code is used in section 1525.
1529. \langle Close the parentheses 1529 \rangle \equiv
  if lParenthCnt > 0 then
     begin gSubexpPtr\uparrow.ProcessRightParenthesis; Accept(sy_RightParenthesis, paRightParenthExp1);
     end
This code is used in section 1525.
1530. Now we have to parse the type subexpression. We basically get the adjectives with GetAdjectiveCluster,
then we get the radix type with RadixTypeSubexpression.
\langle \text{Parse expressions (parser.pas) } 1502 \rangle + \equiv
procedure TypeSubexpression;
  begin gExpPtr\uparrow.CreateSubexpression; gSubexpPtr\uparrow.StartType; gSubexpPtr<math>\uparrow.StartAttributes;
  GetAdjectiveCluster; RadixTypeSubexpression;
  gSubexpPtr\uparrow.CompleteAttributes; gSubexpPtr\uparrow.CompleteType;
  KillSubexpression;
  end;
```

510 TERMS Mizar Parser $\S1531$

```
1531. We should recall from Figure 8 (\S1508) that this is a critical part of parsing terms.
\langle \text{Parse expressions (parser.pas) } 1502 \rangle + \equiv
procedure TermSubexpression;
  var lParenthCnt: integer;
  begin gExpPtr\uparrow.CreateSubexpression; OpenParenth(lParenthCnt); { <math>\S 1503 }
  case CurWord.Kind of
  Identifier, Numeral, LeftCircumfixSymbol, sy\_It, sy\_LeftCurlyBracket, sy\_LeftSquareBracket, sy\_The,
         sy\_Dolar, StructureSymbol:
    begin GetClosedSubterm; CompleteArgument(lParenthCnt); end;
                          \{ skip \} ;
  InfixOperatorSymbol:
  othercases begin gSubexpPtr\uparrow.InsertIncorrTerm; gSubexpPtr\uparrow.FinishArgument;
     WrongWord(paWrongTermBeg);
    end;
  endcases:
  AppendFunc(lParenthCnt); \{ (\S1519) \}
  while lParenthCnt > 0 do \langle Parse arguments to the right 1532 \rangle;
  gSubexpPtr\uparrow.FinishTerm;\ KillSubexpression;
  end:
1532. \langle \text{Parse arguments to the right 1532} \rangle \equiv
  begin ArgumentsTail(MaxVisArgNbr-1); dec(lParenthCnt); qSubexpPtr \uparrow. ProcessRightParenthesis;
  Accept(sy\_RightParanthesis, paRightParenthExp10);
  if CurWord.Kind \neq InfixOperatorSymbol then MissingWord(paFunctExp3);
  AppendFunc(lParenthCnt);
  end
This code is used in section 1531.
1533. This will parse fArgsNbr comma separated terms. It's used to parse the arguments "to the right"
of a term, for parsing the contents of an enumerated set (e.g., \{x, y, z, w\}), among many other places.
  We should recall that the StartArgument method is empty.
\langle \text{Parse expressions (parser.pas) } 1502 \rangle + \equiv
procedure Arguments Tail (fArgsNbr: integer);
  begin while (fArgsNbr > 0) \land Occurs(sy\_Comma) do
    begin qSubexpPtr\uparrow.StartArgument; TermSubexpression; qSubexpPtr\uparrow.FinishArgument;
    dec(fArgsNbr);
    end:
  end;
1534. Attributes, terms, predicates have terms as arguments. This relies upon the FinishArguments
method (\S1405).
\langle \text{Parse expressions (parser.pas) } 1502 \rangle + \equiv
procedure GetArguments(const fArgsNbr: integer);
  begin if fArgsNbr > 0 then
    begin TermSubexpression; gSubexpPtr\uparrow.FinishArgument; ArgumentsTail(fArgsNbr-1);
    end:
  end;
```

 $\S1535$ Mizar Parser FORMULAS 511

Subsection 24.1.2. Formulas

```
1535. Quantified variables looks like
```

```
\langle Variable \rangle \ \{ \text{","} \ \langle Variable \rangle \} \ [("be"|"being") \ \langle Type \rangle]
The parsing routine follows the grammar fairly faithfully.
\langle \text{Parse expressions (parser.pas) } 1502 \rangle + \equiv
procedure QuantifiedVariables;
  begin repeat gSubexpPtr↑.StartQualifiedSegment; ReadWord;
     ⟨ Parse comma-separated variables for quantified variables 1536⟩;
     gSubexpPtr \uparrow. StartQualifyingType;
     if Occurs(sy\_Be) \lor Occurs(sy\_Being) then TypeSubexpression;
     gSubexpPtr \uparrow. FinishQualifiedSegment;
  until CurWord.Kind \neq sy\_Comma;
  end;
1536.
          \langle Parse comma-separated variables for quantified variables 1536\rangle \equiv
  repeat gSubexpPtr\uparrow.ProcessVariable; Accept(Identifier, paIdentExp2);
  until \neg Occurs(sy\_Comma)
This code is used in section 1535.
         The existential formula looks like
                                        ex \langle Quantified\text{-}Variables \rangle \text{ st } \langle Formula \rangle
The parser implements it quite faithfully.
\langle \text{Parse expressions (parser.pas) } 1502 \rangle + \equiv
{\bf procedure}\ {\it Existential Formula};
  begin gSubexpPtr↑.StartExistential; QuantifiedVariables;
  gSubexpPtr\uparrow.FinishQuantified; Accept(sy\_St, paStExp); FormulaSubexpression;
  gSubexpPtr\uparrow.FinishExistential;
  end;
```

512 FORMULAS Mizar Parser $\{1538$

1538. Universally quantified formulas are tricky because both

```
for \langle Quantified\text{-}Variables \rangle holds \langle Formula \rangle
```

and

```
for \langle Quantified\text{-}Variables \rangle st \langle Formula \rangle holds \langle Formula \rangle
```

are acceptable. Furthermore, we may include multiple "for $\langle Quantified\text{-}Variables \rangle$ " (possibly with "st $\langle Formula \rangle$ " restrictions) before arriving at the single "holds $\langle Formula \rangle$ ". The trick is to parse this as

```
for \langle Quantified\text{-}Variables \rangle [st \langle Formula \rangle] [holds] \langle Formula \rangle
```

so the recursive call to parse the final formula enables us to parse another quantified formula.

```
⟨ Parse expressions (parser.pas) 1502⟩ +≡
procedure UniversalFormula;
begin gSubexpPtr↑.StartUniversal; QuantifiedVariables; gSubexpPtr↑.FinishQuantified;
if CurWord.Kind = sy_St then
   begin gSubexpPtr↑.StartRestriction; ReadWord; FormulaSubexpression;
   gSubexpPtr↑.FinishRestriction;
end;
case CurWord.Kind of
sy_Holds: begin gSubexpPtr↑.ProcessHolds; ReadWord end;
sy_For, sy_Ex: ; {fallthrough}
othercases begin gSubexpPtr↑.InsertIncorrFormula; MissingWord(paWrongScopeBeg) end;
endcases;
FormulaSubexpression; gSubexpPtr↑.FinishUniversal;
end;
```

1539. The Parser's current token is either "=" or a predicate symbol. Then we should parse "the right-hand side" of the equation (or formula). The current token's Symbol number is passed as the argument to this procedure.

It's worth recalling the definition of TermBegSys (§1502) which is all the token kinds for starting a term. If the next token is a term, then GetArguments is invoked to parse them.

§1540 Mizar Parser FORMULAS 513

1540. Recall a "multi-predicative formula" is something of the form $a \le x \le b$. More generally, we could imagine the grammar for such a formula resembles:

```
\langle Formula \rangle \{ \langle Multi-Predicate \rangle \langle Term-List \rangle \}
```

The Parser's current token is $\langle Multi-Predicate \rangle$, and we want to keep parsing until the entire multi-predicative formula has been parsed.

We should mention (because I have not seen it discussed anywhere) Mizar allows "does not" and "do not" in formulas (for example, "Y does not overlap X /\ Z"), but Mizar does not support "does" (or "do") without the "not". A 401 error would be raised.

Grammatically, this is known as "do-support", and Mizar uses it for negating predicates. The verb following the "do" is a "bare infinitive" (which is why Mizar allows an "infinitive" for predicates). This makes sense when the predicate uses a "finite verb". For "non-finite verb forms", it is idiomatic English to just negate the verb (as in "Not knowing what that means, I just smile and nod" and "It would be a crime not to learn grammar").

```
\langle \text{Parse expressions (parser.pas) } 1502 \rangle + \equiv
procedure CompleteMultiPredicativeFormula;
  begin qSubexpPtr↑. StartMultiPredicativeFormula;
  repeat case CurWord.Kind of
    sy\_Equal, PredicateSymbol: CompleteRightSideOfThePredicativeFormula(CurWord.Nr);
    sy_Does, sy_Do: \( \text{Parse multi-predicate with "does" or "do" in copula 1541 \( \);
    endcases:
  until \neg (CurWord.Kind \in [sy\_Equal, PredicateSymbol, sy\_Does, sy\_Do]);
  gSubexpPtr \uparrow. FinishMultiPredicativeFormula;
  end:
        ⟨ Parse multi-predicate with "does" or "do" in copula 1541⟩ ≡
1541.
  begin (Consume "does not" or "do not", raise error otherwise 1542);
  if CurWord.Kind \in [PredicateSymbol, sy\_Equal] then
    begin CompleteRightSideOfThePredicativeFormula(CurWord.Nr); gSubexpPtr<math>\uparrow.ProcessNegative; end
  else begin gSubExpPtr \uparrow .InsertIncorrFormula; SynErr(CurPos, paInfinitiveExp) end;
  end
This code is used in section 1540.
1542. (Consume "does not" or "do not", raise error otherwise 1542) \equiv
  gSubexpPtr\uparrow.ProcessDoesNot; ReadWord; Accept(sy\_Not, paNotExpected)
This code is used in section 1541.
```

1543. The Parser is trying to parse a predicate and has just parsed a comma-separated list of terms. Now, the Parser's is either (1) looking at a predicate or equality, or (2) has matched "does not" or "do not" and is now looking at a predicate or equality. In both cases, the Parser tries to complete the formula with the CompletePredicativeFormula procedure.

514 FORMULAS Mizar Parser $\S1544$

```
1544.
\langle \text{Parse expressions (parser.pas) } 1502 \rangle + \equiv
procedure CompleteAtomicFormula(var aParenthCnt : integer);
  var lPredSymbol: integer;
  label Predicate; { not actually used }
  begin (Parse left arguments in a formula 1545);
  case CurWord.Kind of
  sy_Equal, PredicateSymbol: \( \text{Parse equation or (possibly infixed) predicate 1546} \);
  sy_Does, sy_Do: \(\rangle\) Parse formula with "does not" or "do not" 1547\);
  sy_Is: \langle Parse formula with "is not" or "is not" 1548 \rangle;
  othercases begin gSubexpPtr \uparrow. ProcessAtomicFormula; MissingWord(paWrongPredSymbol);
    gSubexpPtr \uparrow . InsertIncorrBasic;
    end;
  endcases;
  end;
1545. \langle \text{Parse left arguments in a formula 1545} \rangle \equiv
  repeat AppendFunc(aParenthCnt);
    if CurWord.Kind = sy\_Comma then
       begin ArgumentsTail(MaxVisArgNbr-1);
       if (aParenthCnt > 0) \land (CurWord.Kind = sy\_RightParanthesis) then
         begin dec(aParenthCnt); gSubexpPtr\uparrow.ProcessRightParenthesis; ReadWord;
         if CurWord.Kind \neq InfixOperatorSymbol then MissingWord(paFunctExp1);
         end;
       end;
  until CurWord.Kind \neq InfixOperatorSymbol
This code is used in section 1544.
1546. \langle Parse equation or (possibly infixed) predicate _{1546}\rangle \equiv
  begin CompletePredicativeFormula(CurWord.Nr);
  if CurWord.Kind \in [sy\_Equal, PredicateSymbol, sy\_Does, sy\_Do] then CompleteMultiPredicativeFormula
  end
This code is used in section 1544.
1547. \langle \text{Parse formula with "does not" or "do not" } 1547 \rangle \equiv
  begin gSubexpPtr \uparrow .ProcessDoesNot; ReadWord; Accept (<math>sy\_Not, paNotExpected);
  if CurWord.Kind \in [PredicateSymbol, sy\_Equal] then
    begin Complete Predicative Formula (CurWord.Nr); qSubexpPtr <math>\uparrow. Process Negative;
    if CurWord.Kind \in [sy\_Equal, PredicateSymbol, sy\_Does, sy\_Do] then
       Complete Multi Predicative Formula\\
    end
  else begin gSubExpPtr↑.InsertIncorrFormula; SynErr(CurPos, paInfinitiveExp) end;
This code is used in section 1544.
```

§1548 Mizar Parser FORMULAS 515

```
1548.
               \langle \text{Parse formula with "is not" or "is not" } 1548 \rangle \equiv
    begin qSubexpPtr↑.ProcessAtomicFormula; ReadWord;
    if (CurWord.Kind = sy\_Not) \land (AheadWord.Kind \in TermBegSys + [ModeSymbol, StructureSymbol, Symbol, StructureSymbol, Symbol, S
                sy\_Set, AttributeSymbol, sy\_Non]) \lor (CurWord. Kind \in TermBegSys + [ModeSymbol,
                StructureSymbol, sy_Set, AttributeSymbol, sy_Non]) then
        begin gSubexpPtr\uparrow.StartType; gSubexpPtr\uparrow.StartAttributes;
        if CurWord.Kind = sy_Not then
            begin gSubexpPtr\uparrow.ProcessNegation; ReadWord; end;
        GetAdjectiveCluster;
        case CurWord.Kind of
        sy\_LeftParanthesis, ModeSymbol, StructureSymbol, sy\_Set: begin RadixTypeSubexpression;
            gSubexpPtr\uparrow.CompleteAttributes;\ gSubexpPtr\uparrow.CompleteType;
            gSubexpPtr\uparrow.FinishQualifyingFormula;
        othercases begin gSubexpPtr\uparrow.CompleteAttributes; gSubexpPtr\uparrow.FinishAttributiveFormula; end;
        endcases;
        end
    else begin qSubExpPtr\uparrow.InsertIncorrFormula; WrongWord(paTypeOrAttrExp); end;
    end
This code is used in section 1544.
1549. There is a comment in Polish, a single word ("Kolejnosc") which translates into English as "Order".
    define starts\_with\_term\_token \equiv Numeral, LeftCircumfixSymbol, sy\_It, sy\_LeftCurlyBracket,
                         sy\_LeftSquareBracket, sy\_The, sy\_Dolar, StructureSymbol
\langle \text{Parse expressions (parser.pas) } 1502 \rangle + \equiv
procedure ViableFormula;
    var lParenthCnt: integer;
    label NotPrivate;
    begin gExpPtr\uparrow.CreateSubexpression; OpenParenth(lParenthCnt);
    case CurWord.Kind of
    sy\_For: UniversalFormula;
    sy\_Contradiction: begin gSubexpPtr\uparrow.ProcessContradiction; ReadWord; end;
    sy\_Thesis: begin gSubexpPtr\uparrow.ProcessThesis; ReadWord; end;
    sy\_Not: begin qSubexpPtr\uparrow.ProcessNot; ReadWord; ViableFormula; KillSubexpression;
        gSubexpPtr \uparrow. ProcessNegative;
        end;
    Identifier: if AheadWord.Kind = sy\_LeftSquareBracket then \langle Parse private formula 1550 \rangle
        else goto NotPrivate;
    starts\_with\_term\_token:
        NotPrivate: begin qSubexpPtr↑.StartAtomicFormula; { ??? TermSubexpression }
            GetClosedSubterm; CompleteArgument(lParenthCnt); CompleteAtomicFormula(lParenthCnt);
    InfixOperatorSymbol, PredicateSymbol, sy\_Does, sy\_Does, sy\_Does, sy\_Equal: begin gSubexpPtr \uparrow .StartAtomicFormula;
        CompleteAtomicFormula(lParenthCnt);
    othercases begin gSubexpPtr\uparrow.InsertIncorrFormula; WrongWord(paWrongFormulaBeq) end;
    endcases; (Close parentheses for formula 1551);
    end;
```

516 FORMULAS Mizar Parser $\S1550$

```
1550.
        \langle \text{ Parse private formula } 1550 \rangle \equiv
  begin qSubexpPtr↑.StartPrivateFormula; ReadWord; ReadWord;
  if CurWord.Kind \neq sy\_RightSquareBracket then GetArguments(MaxVisArgNbr);
  Accept(sy\_RightSquareBracket, paRightSquareExp2); gSubexpPtr \uparrow. FinishPrivateFormula;
  end
This code is used in section 1549.
1551. \langle Close parentheses for formula 1551 \rangle \equiv
  while lParenthCnt > 0 do
    begin Conditional Tail; gSubexpPtr \uparrow. ProcessRightParenthesis;
    Accept(sy\_RightParanthesis, paRightParenthExp4); dec(lParenthCnt); CloseParenth(lParenthCnt);
    end
This code is used in section 1549.
1552. Precedence for logical connectives. We will now "hardcode" the precedence for logical connec-
tives into the Mizar Parser. Negations ("not") binds tighter than conjunction ("&"), which binds tighter
than disjunction ("or"), which binds tighter than implication ("implies" and "iff").
  At this point, for the formula "A & B", the Parser has parsed a formula ("A"), and we want to parse
possible conjunctions. The current token will be "&". If not, then the Parser does nothing: it's "done".
  We will parse conjunction as left associative — so "A & B & C" parses as "(A & B) & C".
\langle \text{Parse expressions (parser.pas) } 1502 \rangle + \equiv
procedure Conjunctive Tail;
  begin while (CurWord.Kind = sy\_Ampersand) \land (AheadWord.Kind \neq sy\_Ellipsis) do
    begin qSubexpPtr\uparrow.ProcessBinaryConnective; ReadWord; ViableFormula; KillSubexpression;
    gSubexpPtr\uparrow.FinishBinaryFormula;
    end:
  end;
1553. Mizar parses flexary conjunctions ("\Phi[0] & ... & \Phi[n]") as weaker than "ordinary conjunction".
For example "\Psi & \Phi[0] & ... & \Phi[n]" parses as "(\Psi & \Phi[0]) & ... & \Phi[n]".
  If the user accidentally forgets the ampersand after the ellipses ("\Phi[0] & ... \Phi[n]"), a 402 error will be
raised.
\langle \text{Parse expressions (parser.pas) } 1502 \rangle + \equiv
procedure FlexConjunctiveTail;
  begin Conjunctive Tail;
  if CurWord.Kind = sy\_Ampersand then
    \mathbf{begin}\ Assert(AheadWord.Kind = sy\_Ellipsis);\ ReadWord;\ ReadWord;\ Accept(sy\_Ampersand, 402);
    qSubexpPtr \uparrow. ProcessFlexConjunction; ViableFormula; ConjunctiveTail; KillSubexpression;
    gSubexpPtr \uparrow . FinishFlexConjunction;
    end:
  end;
```

§1554 Mizar Parser FORMULAS 517

1554. Disjunction binds weaker than flexary conjunction (which binds weaker than ordinary conjunction). As for ordinary conjunction, Mizar parses multiple disjunctions as left associative. So "A or B or C" parses as "(A or B) or C".

```
 \langle \operatorname{Parse\ expressions}\ (\operatorname{parser.pas})\ 1502 \rangle + \equiv \\  \operatorname{procedure\ } DisjunctiveTail; \\  \operatorname{begin\ } FlexConjunctiveTail; \\  \operatorname{while\ } (CurWord.Kind = sy\_Or) \wedge (AheadWord.Kind \neq sy\_Ellipsis) \operatorname{\mathbf{do}} \\  \operatorname{begin\ } gSubexpPtr \uparrow. ProcessBinaryConnective; ReadWord; ViableFormula; FlexConjunctiveTail; \\  KillSubexpression; gSubexpPtr \uparrow. FinishBinaryFormula; \\  \operatorname{\mathbf{end}}; \\  \operatorname{\mathbf{end}}; \\  \operatorname{\mathbf{end}}; \\  \operatorname{\mathbf{end}}; \\ \end{aligned}
```

1555. Parsing a disjunction will have the Parser's current token be "or" only if the next token is an ellipsis ("..."), which is precisely the signal for a flexary disjunction. When the current token is not an "or", then the Parser does nothing (its work is done).

When the user forgets an "or" after ellipsis (e.g., writing "A or ... C"), a 401 error will be raised.

```
 \langle \operatorname{Parse\ expressions}\ (\operatorname{parser.pas})\ 1502 \rangle +\equiv \\ \mathbf{procedure\ } \mathit{FlexDisjunctiveTail}; \\ \mathbf{begin\ } \mathit{DisjunctiveTail}; \\ \mathbf{if\ } \mathit{CurWord.Kind} = \mathit{sy\_Or\ then} \\ \mathbf{begin\ } \mathit{Assert}(\mathit{AheadWord.Kind} = \mathit{sy\_Ellipsis}); \ \mathit{ReadWord}; \ \mathit{ReadWord}; \ \mathit{Accept}(\mathit{sy\_Or\ }, 401); \\ \mathit{gSubexpPtr} \uparrow .\mathit{ProcessFlexDisjunction}; \ \mathit{ViableFormula}; \ \mathit{DisjunctiveTail}; \ \mathit{KillSubexpression}; \\ \mathit{gSubexpPtr} \uparrow .\mathit{FinishFlexDisjunction}; \\ \mathbf{end}; \\ \mathbf{end}; \\ \mathbf{end};
```

1556. Mizar parses "implies" and "iff" with lower precedence than "or", matching common Mathematical practice. Working Mathematicians read "A or B implies C" as "(A or B) implies C". We impose this precedence with the *FlexDisjunctiveTail* parsing the remaining disjunctions before checking for "iff" or "implies".

Mizar accepts one "topmost" implication connective. So "A implies B implies C" would be illegal (a 336 error would be raised). You would have to insert parentheses to make this parseable by Mizar (i.e., "A implies (B implies C)"). This makes sense for implication, but there is a compelling argument that "A iff B iff C" could be parsed as "(A iff B) & (B iff C)" — that latter formula *could* be parsed properly by Mizar.

```
⟨ Parse expressions (parser.pas) 1502⟩ +≡
procedure ConditionalTail;
begin FlexDisjunctiveTail;
case CurWord.Kind of
sy_Implies, sy_Iff: begin gSubexpPtr↑.ProcessBinaryConnective; ReadWord; ViableFormula;
FlexDisjunctiveTail; KillSubexpression; gSubexpPtr↑.FinishBinaryFormula;
case CurWord.Kind of
sy_Implies, sy_Iff: WrongWord(paUnexpConnective);
endcases;
end;
endcases;
end;
```

518 FORMULAS Mizar Parser $\S1557$

1557. Formula subexpressions. When the Parser needs a formula as a subexpression for a larger expression — like when it parses a Fraenkel term (an expression), the Parser will need to parse

```
\{\langle \mathit{Term} \rangle \ \langle \mathit{Qualifying-Segment} \rangle : \langle \mathit{Formula-Subexpression} \rangle \}
```

This will also serve as the "workhorse" for parsing a formula expression.

⟨ Parse expressions (parser.pas) 1502⟩ +≡
procedure FormulaSubexpression;
begin ViableFormula; ConditionalTail; KillSubexpression;
end;

This code is used in section 1493.

Section 24.2. COMMUNICATION WITH ITEMS

1558. When the Parser constructs the AST for a term, the workflow is as follows:

- (1) Allocate a new extExpression object, and update gExprPtr to point at it.
- (2) Using the gExprPtr to allocate a new extSubexp object, and update the gSubexpPtr to point at it.
- (3) The Parser will invoke methods for the gSubexpPtr's reference to build the AST. The result will be stored in a state variable (like gLastTerm or gLastType).
- (4) There will be residual objects allocated, stored in the fields of gSubexpPtr and gExpPtr. We need to clean those up, freeing them, by invoking KillExpression and KillSubexpression.

So each of these methods have the following template: allocate a new expression object, update the gExpPtr to point to it, parse something, then free the gExpPtr using the KillExpression procedure.

```
\langle \mbox{Communicate with items (parser.pas) } 1558 \rangle \equiv \\ \{ \mbox{Communication with items} \} \\ \mbox{procedure } TermExpression; \\ \mbox{begin } gItemPtr \uparrow. CreateExpression(exTerm); TermSubexpression; KillExpression; \\ \mbox{end}; \\ \mbox{procedure } TypeExpression; \\ \mbox{begin } gItemPtr \uparrow. CreateExpression(exType); TypeSubexpression; KillExpression; \\ \mbox{end}; \\ \mbox{procedure } FormulaExpression; \\ \mbox{begin } gItemPtr \uparrow. CreateExpression(exFormula); FormulaSubexpression; KillExpression; \\ \mbox{end}; \\ \mbox{end}; \\ \mbox{end}; \\ \mbox{procedure } TypeSubexpression; \\ \mbox{end}; \\
```

520 MISCELLANEOUS Mizar Parser $\S1559$

Section 24.3. MISCELLANEOUS

1559. When the Parser is looking at a label, the gItemPtr will construct the label. The Parser still needs to move past the " $\langle identifier \rangle$:" two tokens.

```
⟨ Process miscellany (parser.pas) 1559⟩ ≡
   { Miscellaneous }
procedure ProcessLab;
begin gItemPtr↑.ProcessLabel; {(§1365)}
if (CurWord.Kind = Identifier) ∧ (AheadWord.Kind = sy_Colon) then
   begin ReadWord; ReadWord end;
end;
See also sections 1560, 1561, 1562, 1563, 1564, 1565, 1567, 1568, 1569, and 1570.
This code is used in section 1493.
```

1560. Telling the gItemPtr state variable we are about to parse a sentence just invokes the StartSentence (§1280) method, then the Parser parses the formula, and the gItemPtr "finishes" the sentence (which is an empty method).

```
⟨ Process miscellany (parser.pas) 1559⟩ +≡
procedure ProcessSentence;
begin gItemPtr↑.StartSentence; FormulaExpression; gItemPtr↑.FinishSentence;
end:
```

1561. When the Parser expected a sentence but something unexpected happened, specifically an unexpected statement has cross the Parser's path. When that statement has encountered an unjustified "per cases". We just create a new formula expression, and specifically an "incorrect formula".

```
\langle \operatorname{Process\ miscellany\ (parser.pas)\ }^{1559} \rangle + \equiv
procedure InCorrSentence;
begin gItemPtr\uparrow.StartSentence; gItemPtr\uparrow.CreateExpression(exFormula);
gExpPtr\uparrow.CreateSubexpression; gSubexpPtr\uparrow.InsertIncorrFormula; KillSubexpression; KillExpression; gItemPtr\uparrow.FinishSentence;
end;
```

1562. The Parser attempts to recover (or at least, report) an unexpected item when expecting a statement. Specifically, a "per cases" appears when it should not.

```
⟨ Process miscellany (parser.pas) 1559⟩ +≡
procedure InCorrStatement;
begin gItemPtr↑.ProcessLabel; gItemPtr↑.StartRegularStatement; InCorrSentence;
end:
```

§1563 Mizar Parser MISCELLANEOUS 521

1563. The Parser is looking at either

end;

```
let \langle Variables \rangle being \langle Type \rangle such that \langle Hypotheses \rangle
or
                                               assume that \langle Hypotheses \rangle
Specifically, the Parser has arrived at the "\langle Hypotheses \rangle" bit and needs to parse it. The \langle Hypotheses \rangle
generically looks like
                             \langle Hypotheses \rangle = [\langle label \rangle] \langle Formula \rangle \{ and \langle Hypotheses \rangle \}
That is to say, a bunch of (possibly labeled) formulas joined together by "and" keywords.
\langle Process miscellary (parser.pas) 1559 \rangle + \equiv
procedure ProcessHypotheses;
  begin repeat ProcessLab; ProcessSentence; qItemPtr↑.FinishHypothesis;
  until \neg Occurs(sy\_And)
  end;
1564.
         An assumption is either collective (using hypotheses) or singular (a single, possibly labeled, formula).
\langle Process miscellary (parser.pas) 1559 \rangle + \equiv
procedure Assumption;
  begin if CurWord.Kind = sy_That then
     begin gItemPtr↑.StartCollectiveAssumption; ReadWord; ProcessHypotheses
  else begin ProcessLab; ProcessSentence; gItemPtr↑.FinishHypothesis;
     end:
  gItemPtr \uparrow. FinishAssumption;
  end;
1565. Existential elimination in Mizar looks like
                                 consider \langle Fixed\text{-}variables \rangle such that \langle Formula \rangle
The \langle Fixed\text{-}variables \rangle is just a comma-separated list of segments.
\langle Process miscellary (parser.pas) 1559 \rangle + \equiv
procedure FixedVariables;
  begin gItemPtr\uparrow.StartFixedVariables;
  repeat (Parse segment of fixed variables 1566);
  until \neg Occurs(sy\_Comma);
  gItemPtr \uparrow. FinishFixedVariables;
```

522 MISCELLANEOUS Mizar Parser $\S1566$

```
1566. And a "fixed" segment is just a comma-separated list of variables. This is either implicitly qualified (i.e., they are all reserved variables) or explicitly qualified (i.e., there is a "being" or "be", followed by a type). A 300 error will be raised if the comma-separated list of variables encounters something other than an identifier.
```

```
\langle \text{ Parse segment of fixed variables } 1566 \rangle \equiv
  qItemPtr\uparrow.StartFixedSegment;
  repeat gItemPtr↑.ProcessFixedVariable; Accept(Identifier, paIdentExp4);
  until \neg Occurs(sy\_Comma);
  gItemPtr\uparrow.ProcessBeing; { parse the type qualification }
  if Occurs(sy\_Be) \lor Occurs(sy\_Being) then TypeExpression;
  gItemPtr \uparrow . FinishFixedSegment
This code is used in section 1565.
1567. The Parser is trying to parse a "consider" statement or a "given" statement. The Parser will try
to parse
                           \langle Fixed\text{-}Variables \rangle such that \langle Formula \rangle { and \langle Formula \rangle }
If the user forgot the "such" keyword, a 403 error will be raised. If the user forgot the "that" keyword, a
350 error will be raised.
\langle Process miscellany (parser.pas) 1559 \rangle + \equiv
procedure ProcessChoice;
  begin Fixed Variables; Accept(sy\_Such, paSuchExp); Accept(sy\_That, paThatExp2);
  repeat qItemPtr \uparrow .StartCondition; ProcessLab; ProcessSentence; <math>qItemPtr \uparrow .FinishCondition;
  until \neg Occurs(sy\_And):
  gItemPtr \uparrow. FinishChoice;
  end;
         The Parser is looking at the "let" token. There are two possible statements
                                               let \( Fixed-variables \);
or possibly with assumptions
                                 let \langle Fixed\text{-}variables \rangle such that \langle Hypotheses \rangle;
If the user forgot "that" but included a "such" after the fixed-variables, a 350 error is raised.
\langle Process miscellany (parser.pas) 1559 \rangle + \equiv
procedure Generalization;
  begin ReadWord; FixedVariables;
  if Occurs(sy\_Such) then
     begin qItemPtr\uparrow.StartAssumption; Accept(<math>sy\_That, paThatExp1); ProcessHypotheses;
     gItemPtr \uparrow . FinishAssumption;
     end;
  end;
1569. The Parser is looking at the "given" token currently. This is the same as "assume ex \vec{x} st \Phi[\vec{x}];
then consider \vec{x} such that \Phi[\vec{x}];".
\langle Process miscellany (parser.pas) 1559 \rangle + \equiv
procedure Existential Assumption;
  begin qBlockPtr\uparrow.CreateItem(itExistentialAssumption); ReadWord; ProcessChoice;
```

end:

 $\S1570$ Mizar Parser MISCELLANEOUS 523

```
1570. The Parser is looking at either "canceled;" or "canceled ⟨number⟩;".
⟨Process miscellany (parser.pas) 1559⟩ +≡
procedure Canceled;
begin gBlockPtr↑.CreateItem(itCanceled); ReadWord;
if CurWord.Kind = Numeral then ReadWord;
gItemPtr↑.FinishTheorem;
end;
```

524 SIMPLE JUSTIFICATIONS Mizar Parser $\S1571$

Section 24.4. SIMPLE JUSTIFICATIONS

This code is used in section 1572.

1571. The Parser is looking at "by" and now needs to parse the list of references. If the user tries to use something other than a label's identifier as a reference, then a 308 error will be raised.

```
\langle \text{Parse simple justifications (parser.pas) } 1571 \rangle \equiv
    { Simple Justifications }
procedure GetReferences;
  begin gItemPtr\uparrow.StartReferences;
  repeat ReadWord; (Parse single reference 1572);
  until CurWord.Kind \neq sy\_Comma;
  gItemPtr\uparrow.FinishReferences;
  end:
See also sections 1574 and 1576.
This code is used in section 1493.
1572. \langle \text{Parse single reference } 1572 \rangle \equiv
  case CurWord.Kind of
  MMLIdentifier: (Parse library references 1573);
  Identifier: begin gItemPtr↑.ProcessPrivateReference; ReadWord end;
  othercases WrongWord(paWrongReferenceBeg);
  endcases
This code is used in section 1571.
1573. Mizar supports multiple references from the same article to "piggieback" off the same article
"anchor". For example, "GROUP_1:13,def 3,17" refers to theorems 13 and 17 and definition 3 from the
MML Article GROUP_1.
  If the user forgot to include the theorem or definition number — so they just wrote "\( Article \)" instead of
(Article):(Number) or (Article):def(Number) — then Mizar flags this with a 384 error.
  define no\_longer\_referencing\_article \equiv (CurWord.Kind \neq sy\_Comma) \lor
              (AheadWord.Kind = Identifier) \lor (AheadWord.Kind = MMLIdentifier)
\langle \text{ Parse library references } 1573 \rangle \equiv
  begin qItemPtr\\\.StartLibraryReferences; ReadWord;
  if CurWord.Kind = sy\_Colon then
    repeat ReadWord; qItemPtr\\\\.ProcessDef;
       if CurWord.Kind = ReferenceSort then
         begin if CurWord.Nr \neq ord(syDef) then ErrImm(paDefExp);
         ReadWord;
         end:
       gItemPtr\uparrow.ProcessTheoremNumber; Accept(Numeral, paNumExp);
    until no_longer_referencing_article
  else Missing Word (paColonExp4);
  gItemPtr \uparrow . FinishTheLibraryReferences;
  end
```

§1574 Mizar Parser SIMPLE JUSTIFICATIONS 525

1574. The Parser is currently looking at "from", which means a reference to a scheme identifier will be given next (possibly followed with a comma-separated list of references in parentheses).

If the user tries to give something else (instead of an identifier of a scheme), then a 308 error will be raised. Also, if the user forgot the closing parentheses around the references for the scheme (e.g., "from MyScheme(A1,A2"), then 370 error will be raised.

```
\langle \text{Parse simple justifications (parser.pas) } 1571 \rangle + \equiv
procedure GetSchemeReference;
  begin gItemPtr \uparrow .StartSchemeReference; ReadWord;
  case CurWord.Kind of
  MMLIdentifier: (Parse reference to scheme from MML 1575);
  Identifier: begin gItemPtr\uparrow.ProcessSchemeReference; ReadWord end;
  othercases Wrong Word (pa Wrong Reference Beg);
  endcases;
  if CurWord.Kind = sy\_LeftParanthesis then
    begin GetReferences; Accept(sy_RightParanthesis, paRightParenthExp7)
  gItemPtr\uparrow.FinishSchemeReference;
  end;
        Mizar expects scheme references to the MML to be of the form "from \langle Article \rangle:sch \langle Number \rangle". If
the user forgot the "sch" (after the colon), a 313 error will be raised. If the user supplies something other
than a number for the scheme, a 307 error will be raised.
\langle Parse reference to scheme from MML 1575\rangle \equiv
  begin gItemPtr\uparrow.StartSchemeLibraryReference; ReadWord;
  if CurWord.Kind = sy\_Colon then
    begin ReadWord; gItemPtr\(\gamma\).ProcessSch;
    if CurWord.Kind = ReferenceSort then
       begin if CurWord.Nr \neq ord(sySch) then ErrImm(paSchExp);
       ReadWord:
       end
    else ErrImm(paSchExp);
    gItemPtr\uparrow.ProcessSchemeNumber; Accept(Numeral, paNumExp);
    end
  else MissingWord(paColonExp4);
  qItemPtr\uparrow.FinishSchLibraryReferences;
  end
This code is used in section 1574.
1576. The Parser expects a simple justification — i.e., either a "by" followed by some references, or "from"
followed by a scheme reference. For some "obvious" inferences, no justification may be needed.
\langle \text{Parse simple justifications (parser.pas) } 1571 \rangle + \equiv
procedure SimpleJustification;
  begin qItemPtr\uparrow.StartSimpleJustification;
  case CurWord.Kind of
  sy_By: GetReferences;
  sy\_Semicolon, sy\_DotEquals:;
  sy_From: GetSchemeReference;
  othercases Wrong Word (pa Wrong Justification Beg);
  endcases; gItemPtr\uparrow.FinishSimpleJustification;
```

end;

end:

Section 24.5. STATEMENTS AND REASONINGS

1577. Pragmas have been enabled which tells Mizar to skip the proof. The Parser simply stores a counter (initialized to 1), and increments it every time a "proof" token has been encountered, but decrements it every time an "end" token has been encountered. When the counter has reached zero, the proof has ended, and the Parser can stop skipping things.

There are, of course, other blocks which use "end" to terminate it. For example, definitions. But if the Parser should encounter such tokens, then things have gone so horribly awry, the Parser should just quit here and now.

```
\langle \text{Parse statements and reasoning (parser.pas)} | 1577 \rangle \equiv
     { Statements & Reasonings}
procedure Reasoning; forward;
procedure IgnoreProof;
  var lCounter: integer; ReasPos: Position;
  begin qBlockPtr\uparrow.StartAtSiqnProof; ReasPos \leftarrow CurPos; ReadTokenProc; lCounter \leftarrow 1;
  repeat case CurWord.Kind of
    sy\_Proof, sy\_Now, sy\_Hereby, sy\_Case, sy\_Suppose: inc(lCounter);
    sy\_End: dec(lCounter);
    sy\_Reserve, sy\_Scheme, sy\_Theorem, sy\_Definition, sy\_Begin, sy\_Notation, sy\_Registration, EOT: begin
            AcceptEnd(ReasPos); exit
       end:
    endcases; ReadTokenProc;
  until lCounter = 0:
  gBlockPtr\uparrow.FinishAtSignProof;
See also sections 1578, 1579, 1580, 1585, 1586, 1591, 1598, and 1599.
This code is used in section 1493.
1578. Parsing either a "by" justification (or a "from" justification) or a nested "proof" block. If the
Parser is looking at neither situation, the SimpleJustification procedure will raise errors.
  define parse\_proof \equiv
           if ProofPragma then Reasoning
           else IgnoreProof
⟨ Parse statements and reasoning (parser.pas) 1577⟩ +≡
procedure Justification;
  begin qItemPtr \uparrow .StartJustification;
  case CurWord.Kind of
  sy_Proof: parse_proof;
  othercases SimpleJustification;
  endcases; gItemPtr\uparrow.FinishJustification;
```

For private predicates ("defpred") and private functors ("deffunc"), there will be a list of commaseparated types for the arguments of the private definition. **define** $parse_comma_separated_types \equiv$ **repeat** TypeExpression; qItemPtr\(\dagger.FinishLocusType until $\neg Occurs(sy_Comma)$ ⟨ Parse statements and reasoning (parser.pas) 1577⟩ +≡ **procedure** ReadTypeList; begin case CurWord.Kind of $sy_RightSquareBracket$, $sy_RightParanthesis$:; **othercases** parse_comma_separated_types; endcases; end: 1580. A "Private Item" is a statement ("item") which introduces a new constant local ("private") to the block or article. **define** $other_regular_statements \equiv Identifier, sy_Now, sy_For, sy_Ex, sy_Not, sy_Thesis,$ $sy_LeftSquareBracket$, $sy_Contradiction$, PredicateSymbol, sy_Does , sy_Do , sy_Equal , InfixOperatorSymbol, Numeral, LeftCircumfixSymbol, $sy_LeftParanthesis$, sy_It , sy_Dolar , Structure Symbol, sy_The , $sy_Left Curly Bracket$, sy_Proof $\langle \text{Parse statements and reasoning (parser.pas)} 1577 \rangle + \equiv$ **procedure** RegularStatement; forward; $\{(\S1599)\}$ procedure PrivateItem; **begin** $gBlockPtr\uparrow.ProcessLink$; if $CurWord.Kind = sy_Then$ then ReadWord; case CurWord.Kind of $sy_Deffunc: \langle Parse a "deffunc" 1581 \rangle;$ sy_Defpred: \langle Parse a "defpred" 1582 \rangle; sy_Set : $\langle Parse a "set" constant definition 1583 \rangle;$ *sy_Reconsider*: (Parse a "reconsider" statement 1584); $sy_Consider$: **begin** $gBlockPtr\uparrow.CreateItem(itChoice)$; ReadWord; ProcessChoice; SimpleJustification; end: other_regular_statements: **begin** qBlockPtr\.CreateItem(itRegularStatement); RegularStatement; **end**; **othercases begin** *gBlockPtr*↑. CreateItem(itIncorrItem); WrongWord(paWrongItemBeg); **end**; endcases; end; 1581. $\langle Parse a "deffunc" 1581 \rangle \equiv$ **begin** $gBlockPtr \uparrow. CreateItem(itPrivFuncDefinition); ReadWord; <math>gItemPtr \uparrow. StartPrivateDefiniendum;$ $Accept(Identifier, paIdentExp6); Accept(sy_LeftParanthesis, paLeftParenthExp); ReadTypeList;$ $Accept(sy_RightParanthesis, paRightParenthExp8); gItemPtr\uparrow.StartPrivateDefiniens;$ $Accept(sy_Equal, paEqualityExp1); TermExpression; gItemPtr \uparrow. FinishPrivateFuncDefinienition;$ end This code is used in section 1580. 1582. $\langle Parse a "defpred" 1582 \rangle \equiv$ **begin** $qBlockPtr\uparrow.CreateItem(itPrivPredDefinition); ReadWord; <math>qItemPtr\uparrow.StartPrivateDefiniendum;$ $Accept(Identifier, paIdentExp7); Accept(sy_LeftSquareBracket, paLeftSquareExp); ReadTypeList;$ $Accept(sy_RightSquareBracket, paRightSquareExp4); qItemPtr\uparrow.StartPrivateDefiniens;$

 $Accept(sy_Means, paMeansExp); FormulaExpression; gItemPtr \uparrow . FinishPrivatePredDefinienition;$

This code is used in section 1580.

end

```
1583.
        \langle \text{ Parse a "set" constant definition 1583} \rangle \equiv
  begin qBlockPtr\(\frac{1}{2}\). CreateItem(itConstantDefinition); ReadWord;
  repeat qItemPtr \uparrow . StartPrivateConstant; Accept(Identifier, paIdentExp8);
     Accept(sy\_Equal, paEqualityExp2); TermExpression; gItemPtr\uparrow.FinishPrivateConstant;
  until \neg Occurs(sy\_Comma);
  end
This code is used in section 1580.
1584. \langle \text{Parse a "reconsider" statement 1584} \rangle \equiv
  begin gBlockPtr\uparrow.CreateItem(itReconsider); ReadWord;
  repeat qItemPtr \uparrow .ProcessReconsideredVariable; Accept(Identifier, paIdentExp9);
     case CurWord.Kind of
     sy_Equal: begin ReadWord; TermExpression; qItemPtr↑.FinishReconsideredTerm;
     else gItemPtr\uparrow.FinishDefaultTerm;
     end;
  until \neg Occurs(sy\_Comma);
  gItemPtr \uparrow. StartNewType; \ Accept (sy\_As, paAsExp); \ TypeExpression; \ gItemPtr \uparrow. FinishReconsidering; \\
  SimpleJustification;
  end
This code is used in section 1580.
1585. The SetParserPragma toggles the state variables for skipping proofs, and storing the pragma in the
AST is handled by the gBlockPtr's method call.
\langle \text{Parse statements and reasoning (parser.pas)} 1577 \rangle + \equiv
procedure ProcessPragmas;
  begin while CurWord.Kind = Pragma do
     begin SetParserPragma(CurWord.Spelling); \{(\S1195)\}
     gBlockPtr\uparrow.ProcessPragma; \{(\S1215)\}
     ReadTokenProc;
     end;
  end;
        Reasoning items. The "linear reasoning" portion of the parser corresponds to what "Mizar in a
Nutshell" refers to as a sequence of "Reasoning Items". Basically, everything exception "per cases".
\langle \text{ Parse statements and reasoning (parser.pas) } 1577 \rangle + \equiv
procedure LinearReasoning;
  begin while CurWord.Kind \neq sy\_End do
     begin StillCorrect \leftarrow true; ProcessPragmas; \langle Parse statement of linear reasoning 1587 \rangle;
     Semicolon;
     end;
  end;
```

1587. Most statements are delegated to their own dedicated function. $\langle \text{ Parse statement of linear reasoning 1587} \rangle \equiv$ case CurWord.Kind of $sy_Let: \mathbf{begin} \ qBlockPtr\uparrow.CreateItem(itGeneralization); \ Generalization; \mathbf{end};$ sy_Given: ExistentialAssumption; sy_Assume : **begin** $qBlockPtr\uparrow$. CreateItem(itAssumption); ReadWord; Assumption; **end**; sy_Take : $\langle Parse "take" statement for linear reasoning 1588 <math>\rangle$; sy_Hereby : begin $gBlockPtr\uparrow.CreateItem(itConclusion)$; Reasoning; end; (Parse "thus" and "hence" for linear reasoning 1589); $sy_Per: exit:$ $sy_Case, sy_Suppose: exit;$ $sy_Reserve$, sy_Scheme , $sy_Theorem$, $sy_Definition$, sy_Begin , $sy_Notation$, $sy_Registration$, EOT: exit; sy-Then: $\langle Parse$ "then" for linear reasoning 1590 \rangle ; othercases PrivateItem; endcases This code is used in section 1586. Take statements. We recall the syntax for a "take" statement: $take (\langle Term \rangle \mid \langle Variable \rangle = \langle Term \rangle) \{", "(\langle Term \rangle \mid \langle Variable \rangle = \langle Term \rangle)\}$ That is, a comma-separated list of either (1) terms, or (2) a variable equal to a term. $\langle \text{ Parse "take" statement for linear reasoning 1588} \rangle \equiv$ **begin** $qBlockPtr\uparrow$. CreateItem(itExemplification); ReadWord; **repeat if** $(CurWord.Kind = Identifier) \land (AheadWord.Kind = sy_Equal)$ **then begin** $gItemPtr \uparrow .ProcessExemplifying Variable; ReadWord; ReadWord; TermExpression;$ $gItemPtr \uparrow$. FinishExemplifying Variable; else begin $qItemPtr\uparrow.StartExemplifyingTerm; TermExpression; <math>qItemPtr\uparrow.FinishExemplifyingTerm;$ until $\neg Occurs(sy_Comma)$; end This code is used in section 1587. Thus statements. Both "thus" and "hence" (which is syntactic sugar for "then thus") are parsed similarly. So it bears studying them in parallel. The "heavy lifting" is handled by the Regular Statement for parsing the formula. But the gBlockPtr state variable "primes the pump" by creating a "conclusion" statement. $\langle \text{Parse "thus" and "hence" for linear reasoning 1589} \rangle \equiv$ $sy_Hence: \mathbf{begin} \ qBlockPtr\uparrow.ProcessLink; \ ReadWord; \ qBlockPtr\uparrow.CreateItem(itConclusion);$ RegularStatement;end: sy_Thus : **begin** ReadWord; $gBlockPtr\uparrow.ProcessLink$; if $CurWord.Kind = sy_Then$ then ReadWord;

This code is used in section 1587.

 $gBlockPtr\uparrow.CreateItem(itConclusion); RegularStatement;$

```
1590. Parsing 'then' linked statements.
\langle \text{Parse "then" for linear reasoning 1590} \rangle \equiv
  begin if AheadWord.Kind = sy\_Per then
    begin qBlockPtr\uparrow.ProcessLink; ReadWord; exit; end
  else PrivateItem;
  end
This code is used in section 1587.
1591. Non-block Reasoning. The Parser has just encountered a "per cases" statement. Now it must
parse "suppose" items.
⟨ Parse statements and reasoning (parser.pas) 1577⟩ +≡
procedure NonBlockReasoning;
  var CasePos: Position; lCaseKind: TokenKind; \( \rangle \text{Process "case" (local procedure) 1592} \);
  begin case CurWord.Kind of
  sy\_Per, sy\_Case, sy\_Suppose: begin gBlockPtr\uparrow.CreateItem(itPerCases);
    (Consume "per cases", raise an error if they're missing 1593);
    if (CurWord.Kind \neq sy\_Case) \land (CurWord.Kind \neq sy\_Suppose) then
       Try to synchronize after failing to find initial 'case' or 'suppose' 1594);
    repeat (Parse "suppose" or "case" block 1595);
    until (Curword.Kind = sy\_End);
    end;
  endcases;
  end:
1592. Each "case" or "suppose" block consists of zero or more linear reasoning items, followed possibly
by an optional "non-block reasoning" proof (i.e., another nested "per cases" proof by cases).
\langle \text{Process "case" (local procedure) } 1592 \rangle \equiv
procedure ProcessCase;
  begin Assumption; Semicolon; LinearReasoning;
  if CurWord.Kind = sy\_Per then NonBlockReasoning;
  KillBlock; AcceptEnd(CasePos); Semicolon;
  end
This code is used in section 1591.
        The Parser looks for "per cases" tokens, and some simple justification for the statement. If "per"
is missing, a 231 error is raised. If the "cases" is missing, a 351 error is raised. When this code chunk is
done, the Parser is looking at either a "suppose" token or a "case" token.
\langle Consume "per cases", raise an error if they're missing | 1593\rangle \equiv
  Accept(sy\_Per, paPerExp); Accept(sy\_Cases, paCasesExp); SimpleJustification; Semicolon;
  lCaseKind \leftarrow CurWord.Kind
This code is used in section 1591.
```

1594. The Parser is expecting "suppose" or "case" after the "per cases" statement. But if the Parser fails to find either of these tokens, it *should* enter panic mode.

Like a person falling off a cliff reaches out for something to grab, the Parser in panic mode seeks something to "grab on to" so the Parser can "soldier on". The technical term for this situation is that the Parser is trying to "synchronize" (usually people just talk about "synchronization").

Mizar raises a 232 error.

define $create_supposition_block \equiv$

```
⟨ Try to synchronize after failing to find initial 'case' or 'suppose' 1594⟩ ≡
  begin MissingWord(paSupposeOrCaseExp); lCaseKind \leftarrow sy\_Suppose;
  gBlockPtr\uparrow.CreateItem(itCaseBlock);\ gBlockPtr\uparrow.CreateBlock(blSuppose);
  qBlockPtr\uparrow.CreateItem(itSupposeHead); StillCorrect \leftarrow true; CasePos \leftarrow CurPos; ProcessCase;
  end
This code is used in section 1591.
1595. \langle \text{Parse "suppose" or "case" block 1595} \rangle \equiv
  while (CurWord.Kind = sy\_Case) \lor (CurWord.Kind = sy\_Suppose) do
     ⟨ Parse contents of "suppose" block 1596⟩;
  case Curword.Kind of
  sy\_Reserve, sy\_Scheme, sy\_Theorem, sy\_Definition, sy\_Begin, sy\_Notation, sy\_Registration, EOT: exit;
  sy\_End:;
  othercases (Synchronize after missing 'suppose' or 'case' token 1597);
  endcases
This code is used in section 1591.
```

1596. Parsing the contents of a "suppose" or "case" block requires creating a new block (for the, you know, block) and creating a new item for the "suppose (Formula)" or "case (Formula)" statement.

If the user tries to "mix and match" the different kind of suppositions (i.e., "case" and "suppose"), then a 58 error should be raised.

```
if lCaseKind = sy\_Case then gBlockPtr\uparrow.CreateBlock(blCase)
            else gBlockPtr\uparrow.CreateBlock(blSuppose)
  define create\_supposition\_head \equiv
            if lCaseKind = sy\_Case then gBlockPtr \uparrow. CreateItem(itCaseHead)
            else qBlockPtr\(\frac{1}{2}\). CreateItem(itSupposeHead)
⟨ Parse contents of "suppose" block 1596⟩ ≡
  begin qBlockPtr\uparrow.CreateItem(itCaseBlock); create\_supposition\_block; CasePos <math>\leftarrow CurPos;
  StillCorrect \leftarrow true; create\_supposition\_head;
  if CurWord.Kind \neq lCaseKind then ErrImm(58);
  ReadWord; ProcessCase;
  end
This code is used in section 1595.
1597. \langle \text{Synchronize after missing 'suppose' or 'case' token 1597} \rangle \equiv
  begin MissingWord(paSupposeOrCaseExp); gBlockPtr \uparrow. CreateItem(itCaseBlock);
  create\_supposition\_block; create\_supposition\_head; StillCorrect \leftarrow true; CasePos \leftarrow CurPos;
  ProcessCase;
  end
This code is used in section 1595.
```

532

end

This code is used in section 1599.

that we should expect linear reasoning statements, followed by non-block reasoning (i.e., at most one "per cases" statement, and then "suppose" or "case" blocks). ⟨ Parse statements and reasoning (parser.pas) 1577⟩ +≡ procedure Reasoning; var ReasPos: Position; **begin** $ReasPos \leftarrow CurPos$; case CurWord.Kind of sy_Proof : begin $gBlockPtr\uparrow.CreateBlock(blProof)$; ReadTokenProc; end; sy_Hereby : begin $qBlockPtr\uparrow.CreateBlock(blHereby)$; ReadTokenProc; end; sy_Now : begin $gBlockPtr\uparrow.CreateBlock(blDiffuse)$; ReadTokenProc; end; othercases begin $gBlockPtr\uparrow.CreateBlock(blProof)$; WrongWord(paProofExp); end; endcases; LinearReasoning; NonBlockReasoning; KillBlock; AcceptEnd(ReasPos); end; **1599.** Regular statements. A regular statement is one of the following: (1) "now" followed by reasoning; (2) A sentence (i.e., possibly labeled formula) followed by a "proof" block; (3) Iterative equalities. ⟨ Parse statements and reasoning (parser.pas) 1577⟩ +≡ **procedure** RegularStatement; **begin** ProcessLab; gItemPtr↑.StartRegularStatement; case CurWord.Kind of sy_Now: Reasoning; othercases begin *ProcessSentence*; case CurWord.Kind of *sy_Proof*: ⟨Parse "proof" block 1600⟩; **othercases begin** $gItemPtr \uparrow . StartJustification; SimpleJustification; <math>gItemPtr \uparrow . FinishJustification;$ $gItemPtr\uparrow.FinishCompactStatement;$ while $CurWord.Kind = sy_DotEquals$ do $\langle Parse iterative equations 1601 \rangle$; end; endcases: end; endcases; end; **1600.** $\langle \text{Parse "proof" block 1600} \rangle \equiv$ **begin** $qItemPtr\uparrow.StartJustification;$ if ProofPragma then Reasoning **else** *IgnoreProof*; $gItemPtr \uparrow . FinishJustification;$ end This code is used in section 1599. **1601.** $\langle \text{ Parse iterative equations } 1601 \rangle \equiv$ **begin** $gItemPtr\uparrow.StartIterativeStep$; ReadWord; TermExpression; $gItemPtr\uparrow.ProcessIterativeStep$; $qItemPtr \uparrow . StartJustification; SimpleJustification; qItemPtr \uparrow . FinishJustification;$ $gItemPtr\uparrow.FinishIterativeStep$;

Reasoning. The Parser is looking at "proof", "hereby", or "now". The syntax for Mizar says

 $\{1602$ Mizar Parser PATTERNS 533

Section 24.6. PATTERNS

1602. Visible arguments (compared to "hidden arguments") appear to the left or right of a functor or predicate (or to the left of an attribute, or to the right of a mode or structure). The gVisibleNbr state variable is initialized to zero when the Parser starts parsing visible arguments, and the Parser increments it for each visible argument in the pattern.

If a non-identifier appears in a pattern, Mizar raises a 300 error. So you cannot be clever and try to trick Mizar into thinking "0 + x" is a pattern.

```
\langle \text{Parse patterns (parser.pas) } 1602 \rangle \equiv
     { Patterns }
var gVisibleNbr: integer;
procedure GetVisible;
  begin gItemPtr \uparrow .ProcessVisible; \{ (\S 1325) \}
  inc(gVisibleNbr); Accept(Identifier, paIdentExp3);
  end;
See also sections 1603, 1604, 1607, 1608, 1609, 1612, 1614, and 1615.
This code is used in section 1493.
         We will need to Parse a comma-separated list of identifiers when determining a pattern.
\langle \text{Parse patterns (parser.pas) } 1602 \rangle + \equiv
procedure ReadVisible;
  begin gItemPtr \uparrow .StartVisible; gVisibleNbr \leftarrow 0;
  repeat GetVisible;
  until \neg Occurs(sy\_Comma);
  gItemPtr\uparrow.FinishVisible;
  end:
```

1604. There are two cases to consider when determining the pattern for a mode: either the Parser is looking at "set" as a type, or—the more interesting case—the Parser is looking at an identifier which appears in a vocabulary file as a mode symbol.

```
⟨ Parse patterns (parser.pas) 1602⟩ +≡
procedure GetModePattern;
var lModesymbol: integer;
begin gItemPtr↑.StartModePattern; {(§1313)}
case CurWord.Kind of
sy_Set: ⟨Parse pattern for "set" as a mode 1605⟩;
ModeSymbol: ⟨Parse pattern for a mode symbols 1606⟩
othercases WrongWord(paWrongModePatternBeg);
endcases;
gItemPtr↑.FinishModePattern; {(§1314)}
end;

1605. ⟨Parse pattern for "set" as a mode 1605⟩ ≡
begin if AheadWord.Kind = sy_Of then WrongWord(paWrongModePatternSet)
else ReadWord;
end

This code is used in section 1604.
```

534 PATTERNS Mizar Parser $\S1606$

1606. The " $\langle Kind \rangle MaxArgs$ " entry is initialized to \$FF before ReadVisible is invoked, which is PASCAL for "FF = 255. So if the ModeMaxArgs entry for the mode symbol is (1) less than the number of arguments parsed, or (2) uninitialized; then we should update its entry with the gVisibleNbr state variable's current value.

define get_index_compare_to_default(#) \equiv [#] = \$FF

```
define get\_index\_compare\_to\_default(\#) \equiv [\#] = \$FF

define entry\_is\_unitialized(\#) \equiv \#.fList\uparrow get\_index\_compare\_to\_default

\langle Parse \ pattern \ for \ a \ mode \ symbols \ 1606 \rangle \equiv

begin lModeSymbol \leftarrow CurWord.Nr; \ gVisibleNbr \leftarrow 0; \ ReadWord; \ gItemPtr\uparrow.ProcessModePattern;

if Occurs(sy\_Of) then ReadVisible;

if (ModeMaxArgs.fList\uparrow [lModeSymbol] < gVisibleNbr) \lor

(entry\_is\_uninitialized(ModeMaxArgs)(lModeSymbol)) then

ModeMaxArgs.fList\uparrow [lModeSymbol] \leftarrow gVisibleNbr;

end
```

This code is used in section 1604.

1607. Parsing the visible arguments for a functor relies on this helper function.

```
⟨ Parse patterns (parser.pas) 1602⟩ +≡
procedure ReadParams;
begin if Occurs(sy_LeftParanthesis) then
  begin ReadVisible; Accept(sy_RightParanthesis, paRightParenthExp5) end
else if CurWord.Kind = Identifier then
  begin gItemPtr↑.StartVisible; GetVisible; gItemPtr↑.FinishVisible; end;
end;
```

1608. Attribute patterns allows for arguments *only on the right* of the attribute symbol, i.e., something like

```
attr \underbrace{\langle Identifier \rangle \text{ is } \langle Arguments \rangle \ \langle Attribute-Name \rangle}_{\text{pattern}} means...
```

```
\langle \operatorname{Parse \ patterns} \ (\operatorname{parser.pas}) \ 1602 \rangle +\equiv \\ \operatorname{procedure} \ \operatorname{GetAttrPattern}; \\ \operatorname{begin} \ \operatorname{gItemPtr} \uparrow . \operatorname{StartAttributePattern}; \ \operatorname{gVisibleNbr} \leftarrow 0; \ \operatorname{GetVisible}; \\ \operatorname{gItemPtr} \uparrow . \operatorname{ProcessAttributePattern}; \ \operatorname{Accept} (\operatorname{sy\_Is}, \operatorname{paIsExp}); \\ \operatorname{if} \ \operatorname{Occurs} (\operatorname{sy\_LeftParanthesis}) \ \operatorname{then} \\ \operatorname{begin} \ \operatorname{ReadVisible}; \ \operatorname{Accept} (\operatorname{sy\_RightParanthesis}, \operatorname{paRightParenthExp11}) \ \operatorname{end} \\ \operatorname{else} \ \operatorname{if} \ \operatorname{CurWord}. \operatorname{Kind} = \operatorname{Identifier} \ \operatorname{then} \ \operatorname{ReadVisible}; \\ \operatorname{gItemPtr} \uparrow . \operatorname{FinishAttributePattern}; \ \operatorname{Accept} (\operatorname{AttributeSymbol}, \operatorname{paAttrExp2}); \\ \operatorname{end}; \\ \operatorname{end}; \\ \end{aligned}
```

§1609 PATTERNS 535 Mizar Parser

1609. Functor patterns generically look like:

func
$$\underbrace{\langle Arguments \rangle \ \langle Identifier \rangle \ \langle Arguments \rangle}_{\text{pattern}}$$
 -> . . .

or

```
func \langle Left\text{-}Bracket \rangle \langle Arguments \rangle \langle Right\text{-}Bracket \rangle \rightarrow \dots
                                                   pattern
```

```
\langle \text{Parse patterns (parser.pas) } 1602 \rangle + \equiv
procedure GetFuncPattern;
  begin gItemPtr \uparrow .StartFunctorPattern;
  case CurWord.Kind of
  Identifier, InfixOperatorSymbol, sy_LeftParanthesis: \langle Parse infix functor pattern 1610 \rangle;
  LeftCircumfixSymbol, sy_LeftSquareBracket, sy_LeftCurlyBracket: \langle Parse bracket functor pattern 1611 \rangle;
  othercases begin WrongWord(paWrongFunctorPatternBeg); gItemPtr \uparrow .FinishFunctorPattern; end;
  endcases;
  end;
         \langle \text{ Parse infix functor pattern } 1610 \rangle \equiv
  begin ReadParams; gItemPtr\uparrow.ProcessFunctorSymbol; { (§1322) }
  Accept(InfixOperatorSymbol, paFunctExp2); ReadParams; qItemPtr\uparrow.FinishFunctorPattern;
  end
This code is used in section 1609.
1611. \langle \text{Parse bracket functor pattern 1611} \rangle \equiv
  begin ReadWord; ReadVisible; gItemPtr↑.FinishFunctorPattern;
  case Curword.Kind of
  sy\_RightSquareBracket, sy\_RightCurlyBracket, sy\_RightParanthesis: ReadWord;
  othercases Accept(RightCircumfixSymbol, paRightBraExp2);
  endcases:
  end
This code is used in section 1609.
1612. Predicate patterns resemble infix functor patterns.
\langle \text{Parse patterns (parser.pas) } 1602 \rangle + \equiv
procedure GetPredPattern;
  var lPredSymbol: integer;
  begin gItemPtr\uparrow.StartPredicatePattern;
  if CurWord.Kind = Identifier then ReadVisible;
  gItemPtr \uparrow. ProcessPredicateSymbol;
  case CurWord.Kind of
```

 sy_Equal , PredicateSymbol: $\langle Parse predicate pattern 1613 \rangle$;

othercases WrongWord(paWrongPredPattern); endcases; $gItemPtr\uparrow.FinishPredicatePattern$;

end;

536 PATTERNS Mizar Parser $\S1613$

```
1613. \langle \text{Parse predicate pattern } 1613 \rangle \equiv
  begin lPredSymbol \leftarrow CurWord.Nr;
  if CurWord.Kind = sy\_Equal then lPredSymbol \leftarrow EqualitySym;
  qVisibleNbr \leftarrow 0; ReadWord;
  if CurWord.Kind = Identifier then ReadVisible;
  if (PredMaxArgs.fList\uparrow[lPredSymbol] < gVisibleNbr)\lor(entry\_is\_uninitialized(PredMaxArgs)(lPredSymbol))
          then PredMaxArgs.fList \uparrow [lPredSymbol] \leftarrow gVisibleNbr;
  end
This code is used in section 1612.
1614. The "specification" (appearing in a non-expandable mode and functor definitions) refers to the "->
\langle Type \rangle" portion which gives the type for the functor or mode.
\langle \text{Parse patterns (parser.pas) } 1602 \rangle + \equiv
procedure Specification;
  begin gItemPtr \uparrow .StartSpecification; Accept(sy\_Arrow, paArrowExp1); TypeExpression;
  gItemPtr \uparrow . FinishSpecification;
  end:
1615. Parsing a structure pattern is a bit misleading. Unlike the previous procedures, this will actually
parse the entirety of a structure definition:
                                 struct \langle Identifier \rangle ( \langle Types \rangle ) (# \langle Fields \rangle #)
\langle \text{Parse patterns (parser.pas) } 1602 \rangle + \equiv
procedure GetStructPatterns;
  var lStructureSymbol: integer;
  begin gBlockPtr↑.CreateItem(itDefStruct); ReadWord;
  ⟨ Parse ancestors of structure, if there are any 1616⟩;
  ⟨ Parse "over" and any structure arguments, if any 1617⟩;
  gItemPtr \uparrow . StartFields;
  (Update max arguments for structure symbol, if needed 1618);
  ⟨ Parse the fields of the structure definition 1619⟩;
  end;
1616. \langle Parse ancestors of structure, if there are any \frac{1616}{}
  if CurWord.Kind = sy\_LeftParanthesis then
     begin repeat gItemPtr\uparrow.StartPrefix; ReadWord; TypeExpression; gItemPtr\uparrow.FinishPrefix;
     until CurWord.Kind \neq sy\_Comma;
     Accept(sy\_RightParanthesis, paRightParenthExp6);
     end
This code is used in section 1615.
1617. \langle Parse "over" and any structure arguments, if any \frac{1617}{}
  gItemPtr\uparrow.ProcessStructureSymbol;\ lStructureSymbol \leftarrow \$FF;
  if CurWord.Kind = StructureSymbol then lStructureSymbol \leftarrow CurWord.Nr;
  Accept (StructureSymbol, paStructExp1);
  if Occurs(sy_Over) then ReadVisible
This code is used in section 1615.
```

§1618 Mizar Parser PATTERNS 537

```
1618. \langle \text{Update max arguments for structure symbol, if needed 1618} \rangle \equiv
  if lStructureSymbol \neq \$FF then
     \textbf{if} \quad (StructModeMaxArgs.fList \uparrow [lStructureSymbol] \quad < \quad gVisibleNbr) \ \lor \\
             (entry\_is\_uninitialized(StructModeMaxArgs)(lStructureSymbol)) then
        StructModeMaxArgs.fList \uparrow [lStructureSymbol] \leftarrow gVisibleNbr
This code is used in section 1615.
1619. (Parse the fields of the structure definition 1619) \equiv
  Accept(sy_StructLeftBracket, paLeftDoubleExp3);
  repeat (Parse field for the structure definition 1620);
  until \neg Occurs(sy\_Comma);
  gItemPtr\uparrow.FinishFields;\ Accept(sy\_StructRightBracket, paRightDoubleExp2)
This code is used in section 1615.
1620. \langle Parse field for the structure definition _{1620}\rangle \equiv
  qItemPtr\uparrow.StartAqqrPattSeqment;
  repeat gItemPtr↑.ProcessField; Accept(SelectorSymbol, paSelectExp1);
  until \neg Occurs(sy\_Comma);
  Specification;\ gItemPtr \uparrow. FinishAggrPattSegment
This code is used in section 1619.
```

538 DEFINITIONS Mizar Parser $\S1621$

Section 24.7. DEFINITIONS

1621. Non-expandable modes, i.e., modes of the form

```
mode \langle Identifier \rangle of \langle Arguments \rangle \rightarrow \langle Type \rangle means \langle Formula \rangle
\langle \text{Parse definitions (parser.pas) } 1621 \rangle \equiv
     \{ Definitions \}
procedure Construction Type;
  begin gItemPtr\uparrow.StartConstructionType; { (§1364) }
  if CurWord.Kind = sy\_Arrow then
     begin ReadWord; TypeExpression end;
  gItemPtr\uparrow.FinishConstructionType; \{(\S1306)\}
  end:
See also sections 1622, 1623, 1630, 1631, 1632, 1633, 1634, 1638, 1641, 1643, 1647, 1648, 1649, 1650, 1651, 1652, and 1653.
This code is used in section 1493.
1622. Parsing correctness conditions amounts to looping through every "\( Correctness \) \( \) \( Justification \);"
statement, with a fallback "correctness (Justification);" correctness condition.
  There is a comment, "o jaki tu item chodzi? definitional-item?", which Google translates from Polish as,
"What item are we talking about here? Definitional-item?" I have swapped this into the code snippet.
\langle \text{Parse definitions (parser.pas) } 1621 \rangle + \equiv
procedure Correctness;
  begin while CurWord.Kind = sy\_CorrectnessCondition do
     begin StillCorrect \leftarrow true; \ qBlockPtr \uparrow. CreateItem(itCorrCond); \ ReadWord; \ Justification;
     Semicolon;
     end;
  gItemPtr\uparrow.ProcessCorrectness; { (§1363) What item are we talking about here? Definitional-item? }
  if CurWord.Kind = sy_Correctness then { "correctness" catchall }
     begin StillCorrect \leftarrow true; gBlockPtr \uparrow. CreateItem(itCorrectness); ReadWord; Justification;
     Semicolon;
     end;
  end;
```

 $\S1623$ Mizar Parser DEFINITIONS 539

```
1623.
\langle \text{Parse definitions (parser.pas) } 1621 \rangle + \equiv
procedure Definition;
  var lDefKind: TokenKind; lDefiniensExpected: boolean;
  begin lDefKind \leftarrow CurWord.Kind; lDefiniensExpected \leftarrow true;
  case CurWord.Kind of
  sy\_Mode: \langle Parse mode definition 1624 \rangle;
  sy\_Attr: begin gBlockPtr\uparrow. CreateItem(itDefAttr); ReadWord; GetAttrPattern; end;
  sy\_Struct: begin GetStructPatterns; lDefiniensExpected \leftarrow false; end;
  sy\_Func: \mathbf{begin} \ qBlockPtr \uparrow. CreateItem(itDefFunc); \ ReadWord; \ GetFuncPattern; \ ConstructionType;
     end:
  sy\_Pred: begin qBlockPtr\uparrow.CreateItem(itDefPred); ReadWord; qItemPtr\uparrow.StartDefPredicate;
     GetPredPattern;
     end:
  endcases;
  if lDefiniensExpected then \(\rangle\) Parse definiens \(\frac{1625}{2}\rangle\);
  Semicolon; Correctness;
  while (CurWord.Kind = sy\_Property) do
     begin gBlockPtr\uparrow.CreateItem(itProperty); StillCorrect \leftarrow true; ReadWord; Justification; Semicolon;
     end;
  gBlockPtr \uparrow . FinishDefinition;
  end;
1624. \langle Parse mode definition 1624 \rangle \equiv
  begin gBlockPtr↑.CreateItem(itDefMode); ReadWord; GetModePattern;
  case CurWord.Kind of
  sy\_Is: \mathbf{begin} \ qItemPtr \uparrow . StartExpansion; \ ReadWord; \ TypeExpression; \ lDefiniensExpected \leftarrow false;
     end:
  othercases Construction Type;
  endcases;
  end
This code is used in section 1623.
1625. \langle \text{ Parse definiens } 1625 \rangle \equiv
  case CurWord.Kind of
  sy\_Means: \langle Parse "means" definiens 1626 <math>\rangle;
  sy\_Equals: \langle Parse "equals" definiens 1628 <math>\rangle;
  endcases
This code is used in section 1623.
```

540 DEFINITIONS Mizar Parser $\S1626$

```
1626. \langle \text{Parse "means" definiens 1626} \rangle \equiv
  begin qItemPtr\uparrow.ProcessMeans; ReadWord;
  if Occurs(sy\_Colon) then
     begin gItemPtr\uparrow.ProcessDefiniensLabel; Accept(Identifier, paIdentExp10);
     Accept(sy\_Colon, paColonExp2);
  else gItemPtr\uparrow.ProcessDefiniensLabel;
  gItemPtr \uparrow . StartDefiniens; FormulaExpression;
  if CurWord.Kind = sy\_If then \langle Parse "means" definition-by-cases 1627 \rangle
  else gItemPtr\uparrow.FinishOtherwise;
  gItemPtr \uparrow . FinishDefiniens;
  end
This code is used in section 1625.
1627. \langle \text{Parse "means" definition-by-cases 1627} \rangle \equiv
  begin gItemPtr\uparrow.StartGuard; ReadWord; FormulaExpression; gItemPtr\uparrow.FinishGuard;
  while Occurs(sy_Comma) do
     begin Formula Expression; gItemPtr\uparrow. StartGuard; Accept(sy_If, paIfExp); Formula Expression;
     gItemPtr\uparrow.FinishGuard;
     end;
  if CurWord.Kind = sy\_Otherwise then
     begin gItemPtr↑.StartOtherwise; ReadWord; FormulaExpression; gItemPtr↑.FinishOtherwise; end;
  end
This code is used in section 1626.
1628. \langle \text{Parse "equals" definiens 1628} \rangle \equiv
  if lDefKind \neq sy\_Func then
     begin Wrong Word (paUnexpEquals); end
  else begin gItemPtr↑.ProcessEquals; ReadWord;
     if Occurs(sy\_Colon) then
        begin gItemPtr\uparrow.ProcessDefiniensLabel; Accept(Identifier, paIdentExp10);
        Accept(sy\_Colon, paColonExp2);
       end
     else qItemPtr\uparrow.ProcessDefiniensLabel;
     gItemPtr \uparrow . StartEquals; TermExpression;
     if CurWord.Kind = sy\_If then \langle Parse "equals" definition-by-cases 1629 \rangle
     else gItemPtr\uparrow.FinishOtherwise;
     gItemPtr \uparrow . FinishDefiniens;
     end
This code is used in section 1625.
1629. \langle \text{Parse "equals" definition-by-cases 1629} \rangle \equiv
  begin gItemPtr\uparrow.StartGuard; ReadWord; FormulaExpression; gItemPtr\uparrow.FinishGuard;
  while Occurs(sy\_Comma) do
     \textbf{begin} \ \textit{TermExpression}; \ \textit{gItemPtr} \uparrow. \textit{StartGuard}; \ \textit{Accept}(\textit{sy\_If}, \textit{paIfExp}); \ \textit{FormulaExpression};
     gItemPtr\uparrow.FinishGuard;
     end;
  if CurWord.Kind = sy\_Otherwise then
     begin gItemPtr\uparrow.StartOtherwise; ReadWord; TermExpression; gItemPtr\uparrow.FinishOtherwise;
     end;
  end
This code is used in section 1628.
```

 $\S1630$ Mizar Parser DEFINITIONS 541

When introducing a "synonym" or "antonym", the Parser needs to determine what kind of thing is being introduced as a synonym or antonym. This could probably be turned into an case statement, but I am just transcribing the code as faithfully as possible. **define** $is_attr_pattern \equiv (CurWord.Kind = Identifier) \land (AheadWord.Kind = sy_Is)$ **define** $is_infix_pattern \equiv (CurWord.Kind \in [LeftCircumfixSymbol, sy_LeftCurlyBracket,$ $sy_LeftSquareBracket$, $sy_LeftParanthesis$, InfixOperatorSymbol]) $\lor ((CurWord.Kind =$ $Identifier) \land (AheadWord.Kind = InfixOperatorSymbol))$ **define** $is_predicate_pattern \equiv (CurWord.Kind = PredicateSymbol) \lor (CurWord.Kind = sy_Equal) \lor$ $((CurWord.Kind = Identifier) \land (AheadWord.Kind \in [sy_Comma, PredicateSymbol, sy_Equal]))$ **define** $is_selector_pattern \equiv (CurWord.Kind = sy_The) \land (AheadWord.Kind = SelectorSymbol)$ **define** is_forgetful_functor_pattern $\equiv (CurWord.Kind = sy_The) \land (AheadWord.Kind = StructureSymbol)$ $\langle \text{Parse definitions (parser.pas) } 1621 \rangle + \equiv$ **function** CurrPatternKind: TokenKind; **begin** if CurWord.Kind = ModeSymbol then $CurrPatternKind \leftarrow ModeSymbol$ else if CurWord.Kind = StructureSymbol then $CurrPatternKind \leftarrow StructureSymbol$ else if $is_attr_pattern$ then $CurrPatternKind \leftarrow AttributeSymbol$ else if $is_infix_pattern$ then $CurrPatternKind \leftarrow InfixOperatorSymbol$ else if $is_predicate_pattern$ then $CurrPatternKind \leftarrow PredicateSymbol$ else if $is_selector_pattern$ then $CurrPatternKind \leftarrow SelectorSymbol$ else if $is_forgetful_functor_pattern$ then $CurrPatternKind \leftarrow ForgetfulFunctor$ else $CurrPatternKind \leftarrow sy_Error$; end; 1631. The Parser is looking at the "synonym" token when this procedure is invoked. $\langle \text{Parse definitions (parser.pas) } 1621 \rangle + \equiv$ **procedure** Synonym: **begin** ReadWord; case CurrPatternKind of ModeSymbol: begin { Mode synonym } $gBlockPtr\uparrow.CreateItem(itModeNotation);\ GetModePattern;\ gItemPtr\uparrow.ProcessModeSynonym;$ $Accept(sy_For, paForExp); GetModePattern;$ end: AttributeSymbol: begin { Attribute synonym } $gBlockPtr \uparrow. CreateItem(itAttrSynonym); GetAttrPattern; gItemPtr \uparrow. ProcessAttrSynonym;$ $Accept(sy_For, paForExp); GetAttrPattern;$ end; InfixOperatorSymbol: begin { Functor synonym } $qBlockPtr\uparrow.CreateItem(itFuncNotation); GetFuncPattern; qItemPtr\uparrow.ProcessFuncSynonym;$ $Accept(sy_For, paForExp); GetFuncPattern;$ end; PredicateSymbol: begin { Predicate synonym } $qBlockPtr \uparrow. CreateItem(itPredSynonym); \ qItemPtr \uparrow. StartDefPredicate; \ GetPredPattern;$ $gItemPtr\uparrow.ProcessPredSynonym; Accept(sy_For, paForExp); GetPredPattern;$ end

othercases begin $gBlockPtr\uparrow.CreateItem(itIncorrItem); ErrImm(paWrongPattBeq1);$ end;

endcases; end; 542 DEFINITIONS Mizar Parser $\S1632$

Antonyms only make sense for attributes and predicates. A 314 error is raised for any other kind

1632.

of antonym. $\langle \text{Parse definitions (parser.pas) } 1621 \rangle + \equiv$ **procedure** Antonym; **begin** ReadWord; case CurrPatternKind of Attributesymbol: begin { Attribute antonym } $qBlockPtr\uparrow.CreateItem(itAttrAntonym);\ GetAttrPattern;\ qItemPtr\uparrow.ProcessAttrAntonym;$ Accept(sy_For, paForExp); GetAttrPattern; end: PredicateSymbol: begin { Predicate antonym } $qBlockPtr\uparrow.CreateItem(itPredAntonym);\ gItemPtr\uparrow.StartDefPredicate;\ GetPredPattern;$ $gItemPtr\uparrow.ProcessPredAntonym; Accept(sy_For, paForExp); GetPredPattern;$ end othercases begin $qBlockPtr\uparrow.CreateItem(itIncorrItem); ErrImm(paWronqPattBeq2);$ end; endcases: end; 1633. $\langle \text{Parse definitions (parser.pas) } 1621 \rangle + \equiv$ **procedure** *UnexpectedItem*; begin case CurWord.Kind of sy_Case, sy_Suppose, sy_Hereby: **begin** ErrImm(paWronqItemBeq); ReadWord; if $CurWord.Kind = sy_That$ then ReadWord; PrivateItem: end; $sy_Per: \mathbf{begin} \ qBlockPtr\uparrow.CreateItem(itIncorrItem); \ ErrImm(paWronqItemBeq); \ ReadWord;$ if $CurWord.Kind = sy_Cases$ then **begin** ReadWord; InCorrStatement; SimpleJustification; end; end: othercases begin ErrImm(paUnexpItemBeq); $StillCorrect \leftarrow true$; PrivateItem; end; endcases; end; 1634. The Parser is currently looking at the "definition" token, so it will construct a definition block AST. $\langle \text{Parse definitions (parser.pas) } 1621 \rangle + \equiv$ **procedure** DefinitionalBlock; var DefPos: Position; **begin** $qBlockPtr\uparrow.CreateItem(itDefinition); qBlockPtr\uparrow.CreateBlock(blDefinition); DefPos <math>\leftarrow CurPos;$ ReadWord; while $CurWord.Kind \neq sy_End$ do $\langle Parse item in definition block 1635 \rangle$; KillBlock; AcceptEnd(DefPos);end:

 $\{1635$ Mizar Parser DEFINITIONS 543

```
1635.
         \langle \text{ Parse item in definition block 1635} \rangle \equiv
  begin StillCorrect \leftarrow true; gBlockPtr \uparrow .ProcessRedefine;
  if Occurs(sy_Redefine) then (Check we are redefining a mode, attribute, functor, or predicate 1636);
  case CurWord.Kind of
  sy_Mode, sy_Attr, sy_Struct, sy_Func, sy_Pred: Definition;
  sy\_Begin, EOT, sy\_Reserve, sy\_Scheme, sy\_Theorem, sy\_Definition, sy\_Registration, sy\_Notation: break;
  Pragma: ProcessPragmas:
  othercases begin (Parse loci, assumptions, unexpected items in a definition block 1637);
     Semicolon;
     end;
  endcases;
  end
This code is used in section 1634.
1636. Check we are redefining a mode, attribute, functor, or predicate 1636 \ge 100
  if \neg(CurWord.Kind \in [sy\_Mode, sy\_Attr, sy\_Func, sy\_Pred]) then Error(PrevPos, paUnexpRedef)
This code is used in section 1635.
1637.
         \langle \text{ Parse loci, assumptions, unexpected items in a definition block 1637} \rangle \equiv
  case CurWord.Kind of
  sy\_Let: \mathbf{begin} \ gBlockPtr \uparrow. CreateItem(itLociDeclaration); \ Generalization; \mathbf{end};
  sy\_Given: ExistentialAssumption;
  sy\_Assume: begin gBlockPtr\uparrow. CreateItem(itAssumption); ReadWord; Assumption; end;
  sy_Canceled: Canceled:
  sy\_Case, sy\_Suppose, sy\_Per, sy\_Hereby: UnexpectedItem;
  othercases PrivateItem;
  endcases
This code is used in section 1635.
         The Parser's current token is "notation". Notation blocks are very similar in structure to definition
blocks. Unsurprisingly, the Parser's code has a similar structure as parsing a definition block.
\langle \text{Parse definitions (parser.pas) } 1621 \rangle + \equiv
procedure NotationBlock;
  var DefPos: Position;
  begin qBlockPtr\uparrow.CreateItem(itDefinition); qBlockPtr\uparrow.CreateBlock(blNotation); DefPos <math>\leftarrow CurPos;
  while CurWord.Kind \neq sy\_End do \langle Parse item for notation block 1639 \rangle;
  KillBlock; AcceptEnd(DefPos);
  end;
1639. \langle Parse item for notation block 1639\rangle \equiv
  begin StillCorrect \leftarrow true:
  case CurWord.Kind of
  sy\_Begin, EOT, sy\_Reserve, sy\_Scheme, sy\_Theorem, sy\_Definition, sy\_Registration, sy\_Notation: break;
  Pragma: ProcessPragmas;
  othercases (Parse semicolon-separated items in a notation block 1640);
  endcases:
  end
This code is used in section 1638.
```

544 DEFINITIONS Mizar Parser $\S1640$

```
1640.
        \langle Parse semicolon-separated items in a notation block 1640\rangle \equiv
  begin case CurWord.Kind of
  sy\_Synonym: Synonym;
  sy\_Antonym: Antonym;
  sy\_Let: \mathbf{begin} \ gBlockPtr\uparrow.CreateItem(itLociDeclaration); \ ReadWord; \ FixedVariables; \mathbf{end};
  othercases UnexpectedItem;
  endcases:
  Semicolon;
  end
This code is used in section 1639.
1641.
  define ahead\_is\_type \equiv (AheadWord.Kind \in [sy\_Set, ModeSymbol, StructureSymbol))
  define is\_attr\_token \equiv (CurWord.Kind \in [AttributeSymbol, sy\_Non]) \lor
              (CurWord.Kind \in (TermBeqSys - [sy\_LeftParanthesis, StructureSymbol])) \lor
               ((CurWord.Kind = sy\_LeftParanthesis) \land \neg(ahead\_is\_type)) \lor
              (CurWord.Kind = StructureSymbol) \land (AheadWord.Kind = sy\_StructLeftBracket)
\langle \text{Parse definitions (parser.pas) } 1621 \rangle + \equiv
procedure ATTSubexpression(var aExpKind : ExpKind);
  var lAttrExp: boolean;
  begin aExpKind \leftarrow exNull; gSubexpPtr\uparrow.StartAttributes;
  while is\_attr\_token do
     begin gSubexpPtr\uparrow.ProcessNon; lAttrExp \leftarrow CurWord.Kind = sy_Non;
     if CurWord.Kind = sy\_Non then ReadWord;
     ⟨ Parse arguments for attribute expression 1642⟩;
     if CurWord.Kind = AttributeSymbol then
       begin aExpKind \leftarrow exAdjectiveCluster; gSubexpPtr<math>\uparrow.ProcessAttribute; ReadWord; end
     else begin if lAttrExp \lor (aExpKind = exAdjectiveCluster) then
               \{aExpKind = exAdjectiveCluster \text{ is never true }\}
         begin gSubexpPtr \uparrow .ProcessAttribute; SynErr(CurPos, paAttrExp3); end;
       break;
       end;
     end;
  gSubexpPtr \uparrow. CompleteAttributes;
  end:
        \langle Parse arguments for attribute expression 1642 \rangle \equiv
  if (CurWord.Kind \in (TermBegSys - [StructureSymbol])) \lor
          (CurWord.Kind = StructureSymbol) \land (AheadWord.Kind = sy\_StructLeftBracket) then
     begin if aExpKind = exNull then aExpKind \leftarrow exTerm;
     qSubexpPtr\uparrow.StartAttributeArguments;\ ProcessArguments;\ qSubexpPtr\uparrow.FinishAttributeArguments;
     end
This code is used in section 1641.
```

 $\{1643$ Mizar Parser DEFINITIONS 545

```
1643.
         Registration clusters.
\langle \text{Parse definitions (parser.pas) } 1621 \rangle + \equiv
procedure RegisterCluster;
  var lExpKind: ExpKind;
  begin gBlockPtr↑.CreateItem(itCluster); ReadWord;
  if (CurWord.Kind = Identifier) \land (AheadWord.Kind = sy\_Arrow) then ErrImm(paFunctExp4);
  gItemPtr\uparrow.StartAttributes; \{ (\S1278) \}
  gItemPtr\uparrow.CreateExpression(exAdjectiveCluster); \{(\S1351)\}
  gExpPtr\uparrow.CreateSubexpression; ATTSubexpression(lExpKind);
  case lExpKind of
  exTerm: gSubexpPtr \uparrow. CompleteClusterTerm;
  exNull, exAdjectiveCluster: gSubexpPtr \uparrow. CompleteAdjectiveCluster;
  endcases;
  KillSubexpression; KillExpression;
  case lExpKind of
  exTerm: \langle \text{Parse functor registration cluster } 1644 \rangle;
  exNull, exAdjectiveCluster: case CurWord.Kind of
     sy\_Arrow: \langle Parse conditional registration cluster 1645 \rangle;
     sy-For: \langle Parse existential registration cluster 1646 <math>\rangle;
     othercases begin SynErr(CurPos, paForOrArrowExpected); gItemPtr \uparrow .FinishConsequent;
       qItemPtr\uparrow.CreateExpression(exType);\ qExpPtr\uparrow.CreateSubexpression;\ qSubexpPtr\uparrow.StartType;
       gSubexpPtr\uparrow.InsertIncorrType;\ gSubexpPtr\uparrow.CompleteType;\ gSubexpPtr\uparrow.CompleteClusterType;
       KillSubexpression; KillExpression; qItemPtr\uparrow.FinishClusterType;
       end:
     endcases;
  endcases; Semicolon; Correctness;
  end:
1644.
         \langle Parse functor registration cluster 1644\rangle \equiv
  begin gItemPtr\uparrow.FinishClusterTerm; Accept(sy\_Arrow, paArrowExp2);
  qItemPtr \uparrow. CreateExpression(exAdjectiveCluster); qExpPtr \uparrow. CreateSubexpression;
  qSubexpPtr\uparrow.StartAttributes;\ ATTSubexpression(lExpKind);
  if lExpKind \neq exAdjectiveCluster then
     begin ErrImm(paAdjClusterExp) end;
  gSubexpPtr \uparrow. CompleteAdjectiveCluster; \ KillSubexpression; \ KillExpression;
  gItemPtr \uparrow . FinishConsequent;
  if CurWord.Kind = sy\_For then
     begin ReadWord; qItemPtr\uparrow.CreateExpression(exType); qExpPtr\uparrow.CreateSubexpression;
     gSubexpPtr\uparrow.StartType;\ gSubexpPtr\uparrow.StartAttributes;\ GetAdjectiveCluster;\ RadixTypeSubexpression;
     qSubexpPtr\uparrow.CompleteAttributes;\ qSubexpPtr\uparrow.CompleteType;\ qSubexpPtr\uparrow.CompleteClusterType;
     KillSubexpression; KillExpression;
     end;
  gItemPtr \uparrow . FinishClusterType;
  end
This code is used in section 1643.
```

546 DEFINITIONS Mizar Parser $\S1645$

```
1645.
         \langle Parse conditional registration cluster 1645 \rangle \equiv
  begin gItemPtr\uparrow. FinishAntecedent; ReadWord; gItemPtr\uparrow. CreateExpression(exAdjectiveCluster);
  gExpPtr\uparrow.CreateSubexpression;\ gSubexpPtr\uparrow.StartAttributes;\ ATTSubexpression(lExpKind);
  if lExpKind \neq exAdjectiveCluster then
     begin ErrImm(paAdjClusterExp); end;
  gSubexpPtr \uparrow. CompleteAdjectiveCluster; KillSubexpression; KillExpression;
  qItemPtr\uparrow.FinishConsequent; Accept(sy\_For, paForExp); gItemPtr\uparrow.CreateExpression(exType);
  qExpPtr\uparrow.CreateSubexpression;\ qSubexpPtr\uparrow.StartType;\ qSubexpPtr\uparrow.StartAttributes;
  GetAdjectiveCluster;\ RadixTypeSubexpression;\ gSubexpPtr \uparrow. CompleteAttributes;
  qSubexpPtr\uparrow.CompleteType;\ gSubexpPtr\uparrow.CompleteClusterType;\ KillSubexpression;\ KillExpression;
  gItemPtr \uparrow . FinishClusterType;
  end
This code is used in section 1643.
1646. \langle Parse existential registration cluster | 1646\rangle \equiv
  begin gItemPtr\uparrow.FinishConsequent; ReadWord; <math>gItemPtr\uparrow.CreateExpression(exType);
  qExpPtr\uparrow.CreateSubexpression;\ gSubexpPtr\uparrow.StartType;\ gSubexpPtr\uparrow.StartAttributes;
  GetAdjectiveCluster; RadixTypeSubexpression; gSubexpPtr \uparrow. CompleteAttributes;
  qSubexpPtr\uparrow.CompleteType;\ gSubexpPtr\uparrow.CompleteClusterType;\ KillSubexpression;\ KillExpression;
  gItemPtr\uparrow.FinishClusterType;
  end
This code is used in section 1643.
        Reduction registration.
\langle \text{Parse definitions (parser.pas) } 1621 \rangle + \equiv
procedure Reduction;
  var lExpKind: ExpKind;
  begin gBlockPtr \uparrow. CreateItem(itReduction); ReadWord;
  if (CurWord.Kind = Identifier) \land (AheadWord.Kind = sy\_Arrow) then ErrImm(paFunctExp_4);
  qItemPtr \uparrow. StartFuncReduction; TermExpression; qItemPtr \uparrow. ProcessFuncReduction;
  Accept(sy\_To, paToExp); TermExpression; qItemPtr \uparrow. FinishFuncReduction; Semicolon; Correctness;
  end:
1648. Identification registration.
\langle \text{Parse definitions (parser.pas) } 1621 \rangle + \equiv
procedure Identification;
  begin gBlockPtr\uparrow.CreateItem(itIdentify); ReadWord; { begin }
  qItemPtr \uparrow . StartFuncIdentify; GetFuncPattern; qItemPtr \uparrow . ProcessFuncIdentify;
  Accept(sy\_With, paWithExp); GetFuncPattern; gItemPtr\uparrow.CompleteFuncIdentify; { end;}
  if CurWord.Kind = sy_When then
     begin ReadWord;
     repeat gItemPtr\\capp.ProcessLeftLocus; Accept(Identifier, paIdentExp3);
       Accept(sy\_Equal, paEqualityExp1); gItemPtr\uparrow.ProcessRightLocus; Accept(Identifier, paIdentExp3);
     until \neg Occurs(sy\_Comma);
     end:
  Semicolon; Correctness;
  end;
```

§1649 Mizar Parser DEFINITIONS 547

```
1649.
       Property registration.
\langle \text{Parse definitions (parser.pas) } 1621 \rangle + \equiv
procedure RegisterProperty;
  begin qBlockPtr\uparrow. CreateItem(itPropertyRegistration);
  case PropertyKind(CurWord.Nr) of
  sySethood: begin ReadWord; Accept(sy\_of, paOfExp); gItemPtr\uparrow.StartSethoodProperties;
     TypeExpression; gItemPtr \uparrow. FinishSethoodProperties; Justification;
  othercases begin SynErr(CurPos, paStillNotImplemented); end;
  endcases:
  Semicolon;
  end;
1650.
\langle \text{Parse definitions (parser.pas) } 1621 \rangle + \equiv
procedure RegistrationBlock;
  var DefPos: Position;
  \textbf{begin} \ gBlockPtr \uparrow. CreateItem(itDefinition); \ gBlockPtr \uparrow. CreateBlock(blRegistration);
  DefPos \leftarrow CurPos; ReadWord;
  while CurWord.Kind \neq sy\_End do
    begin StillCorrect \leftarrow true;
    case CurWord.Kind of
    sy_Cluster: RegisterCluster;
    sy_Reduce: Reduction;
    sy\_Identify: Identification;
    sy_Property: RegisterProperty;
    sy\_Begin, EOT, sy\_Reserve, sy\_Scheme, sy\_Theorem, sy\_Definition, sy\_Registration, sy\_Notation: break;
    Pragma: ProcessPragmas;
    othercases begin case CurWord.Kind of
       sy\_Let: \mathbf{begin} \ qBlockPtr\uparrow.CreateItem(itLociDeclaration); \ ReadWord; \ FixedVariables; \mathbf{end};
       sy\_Canceled: Canceled;
       sy_Case, sy_Suppose, sy_Per, sy_Hereby: UnexpectedItem;
       othercases PrivateItem;
       endcases;
       Semicolon:
       end;
    endcases;
    end;
  KillBlock; AcceptEnd(DefPos);
  end;
```

548 DEFINITIONS Mizar Parser $\S1651$

```
1651. Reservation.
\langle \text{Parse definitions (parser.pas) } 1621 \rangle + \equiv
procedure Reservation;
  begin qBlockPtr↑. CreateItem(itReservation); ReadWord;
  repeat gItemPtr \uparrow .StartReservationSegment;
     repeat gItemPtr\uparrow.ProcessReservedIdentifier; Accept(Identifier, paIdentExp11);
     until \neg Occurs(sy\_Comma);
     Accept(sy\_For, paForExp);\ gItemPtr\uparrow.CreateExpression(exResType);\ TypeSubexpression;
     KillExpression; gItemPtr \uparrow. FinishReservationSegment;
  until \neg Occurs(sy\_Comma);
  gItemPtr\uparrow.FinishReservation;
  end;
1652.
         Theorem.
\langle \text{Parse definitions (parser.pas) } 1621 \rangle + \equiv
procedure Theorem;
  begin gBlockPtr\uparrow.CreateItem(itTheorem); ReadWord; ProcessLab; <math>gItemPtr\uparrow.StartTheoremBody;
  ProcessSentence;\ gItemPtr \uparrow. FinishTheoremBody;\ Justification;\ gItemPtr \uparrow. FinishTheorem;
  end;
1653.
         Axiom.
\langle \text{Parse definitions (parser.pas) } 1621 \rangle + \equiv
procedure Axiom;
  begin gBlockPtr\uparrow.CreateItem(itAxiom); ReadWord; ProcessLab; <math>gItemPtr\uparrow.StartTheoremBody;
  ProcessSentence;\ gItemPtr\uparrow.FinishTheoremBody;\ gItemPtr\uparrow.FinishTheorem;
  end;
```

 $\{1654$ Mizar Parser SCHEME BLOCKS 549

Section 24.8. SCHEME BLOCKS

```
1654.
\langle \text{Parse scheme block (parser.pas) } 1654 \rangle \equiv
     { Main (with Schemes) }
procedure SchemeBlock;
  var SchemePos: Position;
  begin qBlockPtr\uparrow.CreateItem(itSchemeBlock); qBlockPtr\uparrow.CreateBlock(blPublicScheme); ReadWord;
  qBlockPtr\uparrow.CreateItem(itSchemeHead);\ gItemPtr\uparrow.ProcessSchemeName;\ SchemePos \leftarrow PrevPos;
  if CurWord.Kind = Identifier then ReadWord;
  \langle \text{ Parse scheme parameters 1655} \rangle;
  Accept(sy\_RightCurlyBracket, paRightCurledExp3); gItemPtr\uparrow.FinishSchemeHeading;
  Accept(sy\_Colon, paColonExp3); FormulaExpression; { Scheme-conclusion }
  gItemPtr\uparrow.FinishSchemeThesis; (Parse scheme premises 1656);
  gItemPtr\uparrow.FinishSchemeDeclaration; \langle Parse justification for scheme 1657 \rangle;
  KillBlock;
  end;
This code is used in section 1493.
1655. \langle \text{Parse scheme parameters } 1655 \rangle \equiv
  Accept(sy\_LeftCurlyBracket, paLeftCurledExp);
  repeat gItemPtr \uparrow .StartSchemeSegment;
     repeat gItemPtr\uparrow.ProcessSchemeVariable; Accept(Identifier, paIdentExp13);
     until \neg Occurs(sy\_Comma);
     gItemPtr \uparrow . StartSchemeQualification;
     case CurWord.Kind of
     sy\_LeftSquareBracket: begin ReadWord; ReadTypeList; gItemPtr \uparrow .FinishSchemeQualification;
       Accept(sy\_RightSquareBracket, paRightSquareExp5);
     sy\_LeftParanthesis: begin ReadWord; ReadTypeList; qItemPtr \uparrow .FinishSchemeQualification;
       Accept(sy\_RightParanthesis, paRightParenthExp9); Specification;
     othercases begin ErrImm(paWrongSchemeVarQual); gItemPtr\uparrow.FinishSchemeQualification;
       Specification;
       end:
     endcases; gItemPtr \uparrow . FinishSchemeSegment;
  until \neg Occurs(sy\_Comma)
This code is used in section 1654.
         \langle \text{ Parse scheme premises 1656} \rangle \equiv
  if CurWord.Kind = sy\_Provided then
     repeat gItemPtr↑.StartSchemePremise; ReadWord; ProcessLab; ProcessSentence;
       gItemPtr\uparrow.FinishSchemePremise;
     until CurWord.Kind \neq sy\_And
This code is used in section 1654.
```

550 SCHEME BLOCKS Mizar Parser $\S1657$

```
1657. \langle Parse justification for scheme \frac{1657}{} \rangle \equiv
  if CurWord.Kind = sy\_Proof then
    begin KillItem; { only KillItem which is run outside of Semicolon procedure }
    if \neg ProofPragma then
       begin gBlockPtr↑.StartSchemeDemonstration; IgnoreProof;
       gBlockPtr \uparrow . FinishSchemeDemonstration;
      end
    else begin StillCorrect \leftarrow true; Accept(sy\_Proof, paProofExp);
       gBlockPtr\uparrow.StartSchemeDemonstration;\ LinearReasoning;
      if CurWord.Kind = sy\_Per then NonBlockReasoning;
       AcceptEnd(SchemePos); gBlockPtr\uparrow.FinishSchemeDemonstration;
       end;
    end
  else begin Semicolon;
    if \neg ProofPragma then
       begin gBlockPtr\uparrow.StartSchemeDemonstration; IgnoreProof;
       gBlockPtr \uparrow . FinishSchemeDemonstration;
      end
    else begin StillCorrect \leftarrow true;
      if CurWord.Kind = sy\_Proof then
         begin WrongWord(paProofExp); StillCorrect \leftarrow true; ReadWord;
       gBlockPtr \uparrow. StartSchemeDemonstration; LinearReasoning;
       if CurWord.Kind = sy\_Per then NonBlockReasoning;
       AcceptEnd(SchemePos); gBlockPtr\uparrow.FinishSchemeDemonstration;
       end;
    end
```

This code is used in section 1654.

 $\S1658$ Mizar Parser MAIN PARSE PROCEDURE 551

Section 24.9. MAIN PARSE PROCEDURE

1658. The main *Parse* method essentially skips ahead to the first "begin", then skips ahead to the first top-level block statement.

```
define skip\_to\_begin \equiv ReadTokenProc;
while (CurWord.Kind \neq sy\_Begin) \land (CurWord.Kind \neq EOT) do ReadTokenProc

\langle Main parse method (parser.pas) 1658 \rangle \equiv
procedure Parse;
begin skip\_to\_begin; { Skips ahead until EOT or finds 'begin'}
if CurWord.Kind = EOT then ErrImm(213)
else \langle Parse proper text 1659 \rangle; { CurrWord.Kind = sy\_Begin }
KillBlock;
end;
This code is used in section 1493.
```

1659. Parsing the "text proper" checks that we have encountered a "begin" keyword, then parses the block statements in the article's contents.

Note that ProcessBegin (§1214) and StartProperText (§1216) are both implemented in the extended block class.

```
\langle \text{Parse proper text } 1659 \rangle \equiv  begin gBlockPtr\uparrow.StartProperText; gBlockPtr\uparrow.ProcessBegin; Accept(sy_Begin, 213); while <math>CurWord.Kind \neq EOT do \langle \text{Parse next block } 1660 \rangle; end
```

This code is used in section 1658.

MAIN PARSE PROCEDURE §1660 Mizar Parser

1660. When parsing the next top-level block in a Mizar article, we tell Mizar's parser we are not in "panic mode". Then we test for unexpected "end" tokens. If we can recover a "begin" token, just start the loop

552

```
If we encounter an "end of text" token, then we should terminate the loop.
  Otherwise, we dispatch the parser's control depending on the kind of token we encounter.
\langle \text{ Parse next block 1660} \rangle \equiv
  begin (Parse pragmas and begins 1661);
  StillCorrect \leftarrow true;  { we are not in panic mode }
  if CurWord.Kind = sy\_End then
    begin (Skip all end tokens, report errors 1662);
    if CurWord.Kind = sy\_Begin then continue;
  if CurWord.Kind = EOT then break;
  case CurWord.Kind of
  sy\_Scheme: SchemeBlock;
  sy_Definition: DefinitionalBlock;
  sy_Notation: NotationBlock;
  sy\_Registration: RegistrationBlock;
  sy\_Reserve: Reservation;
  sy\_Theorem: Theorem;
  sy\_Axiom: Axiom;
  sy_Canceled: Canceled;
  sy\_Case, sy\_Suppose, sy\_Per, sy\_Hereby: UnexpectedItem;
  othercases PrivateItem;
  endcases;
  Semicolon; { block is expected to end in a semicolon }
This code is used in section 1659.
        The ProcessPragmas (§1585) consumes a token when the current token is a pragma. So we effectively
have a loop where we consume all the pragmas and the "begin" keywords until we find something else.
\langle \text{ Parse pragmas and begins 1661} \rangle \equiv
  while CurWord.Kind \in [sy\_Begin, Pragma] do
    begin ProcessPragmas;
    if CurWord.Kind = sy\_Begin then
       begin gBlockPtr\\\\\?. ProcessBegin; ReadTokenProc;
    end
This code is used in section 1660.
1662. In the unfortunate event that the parser has stumbled across an "end" token, skip all the "end"
and semicolon tokens and report errors.
\langle Skip all end tokens, report errors 1662 \rangle \equiv
  repeat ErrImm(216); ReadTokenProc;
    if CurWord.Kind = sy\_Semicolon then ReadTokenProc;
  until CurWord.Kind \neq sy\_End
This code is used in section 1660.
```

 $\S1663$ Mizar Parser DEFERRED 553

1663. Deferred.

This will not be analyzed until first_identification.pas \langle Class for Within expression 1663 $\rangle \equiv$ This code is used in section 737.

1664. \langle Within expression AST implementation $1664 \rangle \equiv$ This code is used in section 736.

1665. Index. Underlined entries in an index item refers to which section defines the identifier. Primitive types (*char*, *Boolean*, *string*, etc.) are omitted from the index.

```
.frt file:
           850.
                                                      AChar: 533, 1148.
.idx file: 733.
                                                      aChar: 30, 32, 538, 1150.
.prf File: 727.
                                                      aCluster: 804, 805, 1014, 1020, 1148, 1156.
.wsx file: 850.
                                                      aCode: 195, 208, 429, 430, 447, 448.
::$EOF: 614.
                                                      a Comment: 57, 71.
a: 137.
                                                      aCompare: 250, 251, 257, 258, 259, 267, 268,
aAdjectiveNr: 751, 752.
                                                           286, 287.
aAdjectives: 816, 817.
                                                      aConcl: 888, 889.
aAncestors: 930, 931.
                                                      aCond: 487, 492, 1148, 1181, 1182.
aAnother: 195, 198, 199, 206, 219, 230, 233, 234,
                                                      aConds: 906, 907, 982, 983.
    250, 252, 253, 309, 310.
                                                      aCons: 987, 988, 989, 990, 991, 992, 993, 994.
AAnother: 219, 230.
                                                      aCStm: 1014, 1075, 1148, 1172.
aAntec: 991, 992.
                                                      aDctFileName: 593, 596, 599, 600, 632, 645,
aApplicationName: 57, 71.
                                                           723, 730, 732.
aArg: 749, 750, 770, 771, 772, 773, 778, 779,
                                                      Add: 129, 143, 147, 154, 155, 158, 161, 379,
    822, 823, 937, 938.
                                                           394, 429, 445.
                                                      Add Ext Items \colon \ \ 235, \ 247, \ 267, \ 280.
aArgs: 751, 752, 764, 765, 766, 767, 768, 769, 776,
    777, 798, 799, 800, 801, 802, 803, 818, 819,
                                                      AddExtObject: 235, 246, 267, 274, 279, 280.
    935, 936, 937, 938, 941, 942.
                                                      AddObject: 309, 315, 316, 583.
aArgsNbr: 657, 658, 665, 666, 668, 671, 673.
                                                      addr: 301, 305.
aArticleExt: 856, 857.
                                                      Address of: 194.
aArticleID: 856, 857.
                                                      AddSequence: 402, 405, 409.
aArticleNr: 871, 872, 881, 882.
                                                      AddString: 309, 312, 313, 315, 327.
aAttr: 1014, 1019, 1148, 1155.
                                                      AddStrings: 309, 316.
aAttrName: 528, 530, 531, 532, 1097, 1099.
                                                      aDef: 950, 951, 960, 961, 962, 963, 964, 965, 966,
aBase: 18, 1199, 1393.
                                                           967, 1014, 1068, 1148, 1186.
aBlock: 1014, 1074, 1075, 1076, 1148, 1172,
                                                      aDefKind: 956, 957.
                                                      a DefNr: 871, 872.
    1174, 1180.
aBlockKind: 856, 858.
                                                      aDefWay: 966, 967.
aBlokKind: 856, 859.
                                                      aDelta: 287, 359, 360.
                                                      ADelta: 219, 220, 286, 287, 360, 379, 380, 397, 398.
Abs: 129, <u>138</u>, 165.
abs: 140, 152, 153, 154, 155, 156, 157, 159.
                                                      aDir: 34, 38.
abstract\_syntax\colon \quad \underline{735},\ 850,\ 1199.
                                                      AdjectiveExpressionObj: 747, 748, 749, 751.
Abstract1: 183, 288, 308, 461, 628.
                                                      AdjectiveExpressionPtr: 747, 749, 750, 1014, 1019,
AbsVocabularyObj: 571, 572, 573.
                                                           1097, 1108, 1148, 1155, 1394.
AbsVocabularyPtr: 571.
                                                      AdjectiveObj: 751, 752.
                                                      AdjectivePtr: 751, 1019, 1108, 1155, 1395.
aCapacity: 309, 310, 335, 402, 404, 417.
Accept: 1500, 1503, 1507, 1509, 1510, 1511, 1513,
                                                      AdjectiveSort: 747, 748.
    1514, 1516, 1517, 1519, 1522, 1529, 1532, 1536,
                                                      Adjective SortName: 1108.
    1537, 1542, 1547, 1550, 1551, 1553, 1555, 1566,
                                                      aElem: 487, 493.
    1567, 1568, 1573, 1574, 1575, 1581, 1582, 1583,
                                                      aEnvname: 44, 45.
    1584, 1593, 1602, 1607, 1608, 1610, 1611, 1614,
                                                      aEnvName: 44.
    1616, 1617, 1619, 1620, 1626, 1627, 1628, 1629,
                                                      aEqList: 997, 998.
    1631, 1632, 1644, 1645, 1647, 1648, 1649, 1651,
                                                      aErr: 487, 493, 502, 505.
    1654, 1655, 1657, 1659.
                                                      aErrorNbr: 59, 76.
AcceptEnd: 1499, 1577, 1592, 1598, 1634, 1638,
                                                      aExp: 958, 959.
    1650, 1657.
                                                      aExpKind: 18, 1641, 1642.
                                                      aExpr: 948, 949.
AcceptEndState: \underline{502}, 510, 676, 677.
AcceptStartState: 502, 510.
                                                      aExt: 34, 38, 84, 87.
Accomodation: 62, 68.
                                                      aFields: 926, 927, 930, 931.
```

aFieldSymbNr: 924, 925. AItem: 235, 242, 267, 276. aFileExt: 34, 43, 47, 49, 50. aItem: 195, 204, 205, 209, 219, 227, 235, 238, 250, 254, 255, 264, 267, 274, 286, 289, 290, aFileName: 34, 35, 37, 38, 39, 40, 41, 42, 43, 46, 48, 49, 494, 495, 496, 502, 503, 557, 341, 345, 350, 352, 353, 359, 363, 429, 433, 447, 451, 1097, 1142, 1143. 573, 576, 580, 583, 584, 585, 632, 645, 723, 730, 732, 1014, 1015, 1096, 1097, 1098, 1147, aItemKind: 856, 858, 860, 861. aItems: 347, 435. 1148, 1149, 1192. AFileName: 520, 522, 524, 526, 528, 529, 533, AItems: 347, 435. 534, 535, 1149. aItem1: 250, 647, 659, 660, 661, 662, 663, 664.AFilename: 534.aItem2: 250, 647, 659, 660, 662, 663, 664. aFormat: 665, 670. aIters: 916, 918. aFormNr: 651, 678. aJustification: 900, 901, 906, 907, 913, 914, *aFormula*: 784, 785. 915, 916, 917, 918, 978, 979, 980, 981, 982, aFrm: 1014, 1026, 1027, 1028, 1029, 1030, 1031, 983, 984, 985. 1032, 1033, 1034, 1035, 1036, 1037, 1038, 1039, ak: 1426, 1428. 1040, 1041, 1148, 1163, 1164, 1165. aKey: 250, 260, 261, 262, 263, 267, 270, 271, 272. AfterBalance: 1414, 1417, 1418, 1419, 1420, aKey1: 282, 390, 461. 1423, 1424, 1425. aKey2: 282, 390, 461. aFunc: 350, 355, 379, 394, 395, 429, 438, 445, aKind: 591, 592, 601, 602, 651, 652, 653, 654, 655, 447, 456. 656, 665, 666, 673, 764, 765, 770, 771, 798, 799, 900, 901, 941, 942, 946, 947, 948, 949, 987, 988, aFuncId: 893, 894. aFunctorIdNr: 768, 769. 999, 1000, 1001, 1002, 1193, 1194. aFunctorNr: 762, 763. alab: 865. aFunc1: 379, 381. aLab: 864, 865, 911, 912, 946, 947, 950, 951, 954, aFunc2: 379, 381. 955, 1014, 1067, 1148, 1170. Aggregate Term Obj: 776, 777.aLabelId: 864, 865. Aggregate TermPtr: 776, 1049, 1121, 1168, 1445. aLabId: 868, 869. aGuard: 952, 953. aLArgs: 939, 940, 941, 942. aLArgsNbr: 655, 656, 657, 658, 665, 667, 668, $ahead_is_attribute_argument$: 1524. 669, 671, 672, 674. $ahead_is_type$: 1641. AheadPos: 723, 726, 729. aLBSymb: 941, 942. AheadWord: 723, 726, 729, 1237, 1239, 1362, 1365, aLeft: 267, 272. aLeftArg: 824, 825, 826, 827, 828, 829, 830, 831, 1368, 1408, 1446, 1448, 1509, 1516, 1517, 1524, 1548, 1549, 1552, 1553, 1554, 1555, 1559, 1573, 832, 833, 834, 835, 836, 837. aLeftArgs: 762, 763, 812, 813. 1588, 1590, 1605, 1630, 1641, 1642, 1643, 1647. aID: 53. aLeftBracketNr: 766, 767. aIdentifier: 743, 744. aLeftLocus: 995, 996.aldentifiers: 745, 746, 890, 891. a Length: 24, 26.aIdentNr: 738, 739, 756, 757, 928, 929. aLexem: 593, 595. aLimit: 196, 197, 235, 250, 251, 256, 267, 268, aIdNr: 888, 889, 1013. 277, 287, 341, 343, 347, 350, 351, 359, 360, 361, aIndex: 195, 205, 207, 213, 235, 244, 245, 250, 429, 431, 435, 447, 449, 453, 461. 254, 257, 260, 261, 262, 263, 266, 267, 270, ALimit: 195, 215, 219, 220, 235, 236, 243, 250, <u>271</u>, 273, 278, <u>309</u>, 318, <u>320</u>, 322, 323, 327, 328, 330, 332, 333, 341, 346, 350, 353, 354, 251, 267, 268, 277, 281, 283, 286, 287, 347, 360, 379, 380, 397, 398, 435, 665, 675. 402, 411, 413, 415, 416, 418, 421, 423, 429, 434, 437, 447, 452, 455. aLine: 558, 562, 564, 1193, 1196, 1197. aLinked: 879, 880. aInf: 1014, 1072, 1073, 1074, 1148, 1177, 1178, aLinkPos: 879, 880. 1179, 1180. aInferSort: 875, 876, 877, 878. aList: 257, 403. aInfo: 195, 208, 429, 430, 447, 448. Allowed: 53, 632, 644, 645, 726. aInt: 30, 33, 402, 408, 414, 415, 416, 418, 419, aLoci: 1014, 1060, 1148, 1184. 420, 421, 422, 423, 424. aLocus: 1014, 1059, 1148, 1183.

aLocusNr: 758, 759. aPostqual: 782, 783, 784, 785. append: 109.aLSymNr: 657, 658, 665, 668, 671. aMmlVcb: 557, 583, 584, 585. $append_dir_separator$: 482, 483. aModeSymbol: 800, 801.AppendErrors: 105, 109. AnalyzerOnly: 64, 69. AppendFunc: <u>1519</u>, 1521, 1531, 1532, 1545. aName: 34, 38, 59, 72, 490, 491, 1005, 1006. AppendQua: 1504, 1518.AppendTo: 195, 216.aName1: 34, 36.aName2: 34, 36.aPredId: 895, 896. aPredIdNr: 818, 819. Anaphora: 792. aNbr: 1193, 1194.aPredNr: 810, 811, 812, 813. aNewPatt: 968, 969, 997, 998. *aPredSymbol*: 1539, 1543. aPrems: 888, 889. aNewTerm: 1003, 1004.aNr: 591, 592, 601, 602, 881, 882. aPrfFileName: 723, 727. aPrg: 1193, 1194, 1195, 1196. AnsiChar: 538, 596, 599, 1008. aProgName: 84, 87.aNSort: 968, 969. aProp: 913, 914, 916, 918, 972, 973, 1014, 1069, Antonym: 1632, 1640. 1148, 1171. aNumber: 1148, 1152. aObject: 309, 315, 329, 330, 333. aProps: 974, 975, 976, 977. aQualVars: 906, 907.aOrigPatt: 968, 969, 997, 998. aQVars: 976, 977. aOrigTerm: 1003, 1004.aRArgs: 939, 940, 941, 942. aOtherwise: 954, 955. aRArgsNbr: 653, 654, 655, 656, 657, 658, 665, aOverArgs: 930, 931. aParams: 888, 889. 667, 668, 669, 671, 672, 674. aRBSymb: 941, 942.aParenthCnt: 1544, 1545.aRedef: 956, 957, 960, 961, 962, 963, 964, aPartDef: 952, 953.965, 966, 967. aPartialDefs: 954, 955. AreEqComplex: 133, 170.aPassName: 84, 85.aRef: 1014, 1070, 1148, 1175. aPattern: 956, 957, 958, 959, 960, 961, 962, 963, aRefs: 1014, 1071, 1148, 1176. 964, 965, 966, 967, 1014, 1061, 1062, 1063, aRes: 1014, 1077. 1064, 1065, 1066, 1148, 1185. aResType: 1148, 1189.aPos: 601, 602, 738, 739, 741, 742, 743, 744, 745, arg_type: 1387, 1388. 746, 747, 748, 749, 750, 751, 752, 756, 757, 758, ArgsLength: 1417, 1418, 1420, 1421, 1422, 1423, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 1424, 1425. 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, Arguments Tail: 1506, 1514, 1521, 1532, 1533, 779, 780, 781, 782, 783, 784, 785, 786, 787, 788, 1534, 1545. 789, 790, 791, 792, 793, 794, 795, 798, 799, 800, aRight: 267, 272. 801, 802, 803, 804, 805, 806, 807, 810, 811, 812, aRightArg: 824, 825, 826, 827, 828, 829, 830, 831, 813, 814, 815, 816, 817, 818, 819, 820, 821, 822, 823, 824, 825, 826, 827, 828, 829, 830, 831, 832, 832, 833, 834, 835, 836, 837. aRightArgs: 762, 763, 810, 811, 812, 813. 833, 834, 835, 836, 837, 838, 839, 840, 841, 842, aRightBracketNr: 766, 767. 843, 844, 845, 846, 847, 848, 849, 856, 857, 858, aRightLocus: 995, 996. 859, 860, 861, 864, 865, 868, 869, 871, 872, 875, aRStm: 1014, 1076, 1148, 1174. 876, 877, 878, 879, 880, 881, 882, 884, 885, 886, 887, 888, 889, 915, 917, 924, 925, 926, 927, 928, aRSymNr: 657, 658, 665, 668, 671. ArticleExt: 24, 50, 69, 482, 1203. 929, 930, 931, 933, 934, 935, 936, 937, 938, 939, ArticleID: 24, 50, 600, 1203. 940, 941, 942, 946, 947, 950, 951, 954, 955, 956, 957, 958, 959, 960, 961, 962, 963, 964, 965, 966, ArticleName: 24, 50, 482. 967, 968, 969, 970, 971, 972, 973, 974, 975, 976, aSample: 782, 783, 784, 785. aSch: 1014, 1078, 1148, 1190. 977, 978, 979, 980, 981, 982, 983, 984, 985, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, AsciiArr: 590. 998, 999, 1000, 1001, 1002, 1003, 1004. ASCIIArr: 589, 632, 633. Apos: 773. aScope: 838, 839, 840, 841, 842, 843.

aScraps: 814, 815. atArticleID: 1144. aSeqm: 1014, 1024, 1025, 1148, 1158, 1159. atArticleId: 462, 1016. aSegment: 838, 839, 840, 841, 842, 843. atCol: 462, 552, 1100, 1144, 1145, 1146. aSegmSort: 884, 885. atCondition: 462, 1079, 1087, 1142, 1143. aSelectorNr: 772, 773, 774, 775. at ConstrNr: 462. aSentence: 864, 865. AtDelete: 219, 221, 222, 225, 228, 341, 346, 370, $aSeq: \underline{402}, 405, \underline{406}, 409.$ 372, 384, 402, 411, 420, 429, 434, 443, 447, 452. aTerm: 893, 894, 897, 898, 900, 901, 903, 904, aSet: 418, 425, 426, 427, 428. aShape: 1097, 1141, 1142, 1143. 915, 917. aSig: 95.aText: 562, 568, 569, 573, 578, 579, 580.aSignal: 100, 101.AtFree: 219, 222, 630. aSnt: 895, 896. a Theo Nr: 871, 872.aSntPos: 864, 865. atIdNr: 462, 1022, 1026, 1042, 1043, 1059, 1067, aSort: 741, 742, 747, 748, 933, 934, 970, 971, 1070, 1073, 1078, 1111, 1114, 1118, 1119, 1123, 980, 981, 984, 985. 1131, 1133, 1136, 1139. aSpec: 926, 927, 960, 961, 966, 967. aTime: 59, 72, 120. aSpecification: 886, 887. AtIndex: 267, 273. aSpelling: 591, 592, 593, 595, 601, 602. atInfinitive: 462, 587. Assert: 1553, 1555. AtInsert: 195, 205, 219, 223, 227, 250, 254, 255, assign: 596, 599, 645, 727, 1009, 1012. 290, 291, 350, 353, 394, 402, 415, 418, 423. Assign: 46, 108, 109, 379, 381, 382, 429, 441, 445, atKind: 462, 586, 587, 600, 676, 677, 678, 679, 1017, 1068, 1095, 1130, 1145, 1146. 447, 459, 472, 473, 475, 476, 477, 478, 480, 486, 496, 522, 526, 576, 583, 584, 676, 1197. at Label Nr: 462.Assigned: 270. atLeftArgNr: 462, 677, 678.AssignPair: 341, 349, 350, 358. atLine: 462, 552, 1100, 1144, 1145, 1146. Assumption: 1564, 1587, 1592, 1637. atMizfiles: 462, 600. AssumptionKind: 970, 971. atName: 462, 585, 587, 600. atNegated: 462.AssumptionKindName: 1092, 1143.AssumptionObj: 970, 971, 972, 974, 977. atNr: 462, 586, 600, 678, 1019, 1021, 1035, 1036, AssumptionPtr: 970, 1092, 1148, 1182, 1191. 1044, 1046, 1047, 1048, 1049, 1050, 1051, 1052, aStatementSort: 909, 910, 911, 912. 1062, 1063, 1064, 1065, 1066, 1071, 1073, 1091, 1108, 1110, 1117, 1120, 1121, 1125, 1126, 1127, aStr: 30, 31, 32, 57, 71, 191, 192, 281, 284, 285, 309, 312, 313, 315, 320, 325, 326, 327, 1128, 1134, 1139, 1143. 328, 330, 332, 514, 516, 519, 862, 863, 1103, atNumber: 462, 1045, 1071, 1121, 1134. 1104, 1105, 1106. aToken: 604, 605, 606, 607. AString: 533. at Origin: 462. aString: 24, 26, 27, 28, 29, 541, 1148, 1153. atPosCol: 462, 1017, 1073, 1095, 1139, 1145, 1146. aStrings: 309, 316. atPosLine: 462, 1017, 1073, 1095, 1139, 1145, aStructSymb: 930, 931. 1146. aStructSymbol: 776, 777, 778, 779, 780, 781, atPriority: 462, 587. atProperty: 462, 1079, 1142. 802, 803. aStr1: 296, 461. AtPut: 219, 224, 402, 416. aStr2: 296, 461. atRightSymbolNr: 462, 677, 678. aSubject: 786, 787, 788, 789, 816, 817, 820, 821. aTrm: 993, 994, 1014, 1042, 1043, 1044, 1045, aSym: 502, 505. 1046, 1047, 1048, 1049, 1050, 1051, 1052, 1053, aSymb: 935, 936, 937, 938, 939, 940, 941, 942. 1054, 1055, 1056, 1057, 1148, 1166, 1167, 1168. aSymNr: 651, 652, 653, 654, 655, 656, 665, 666, aTrmList: 1014, 1018, 1148, 1160, 1161. 667, 669, 672, 673, 674. atSchNr: 462. At: 195, 202, 203, 222. at Serial Nr: 462.atAid: 462, 600. atShape: 462, 1068, 1079, 1130, 1142. atArgNr: 462, 677, 678. atSpelling: 462, 1019, 1021, 1022, 1026, 1035, atArticleExt: 462, 1016, 1144. 1036, 1042, 1043, 1044, 1046, 1047, 1048, 1049,

1050, 1051, 1059, 1062, 1063, 1064, 1065, 1066, *aXMLStr*: 514, 517. 1067, 1070, 1071, 1073, 1078, 1079, 1091, 1142. a1: 137, 143, 144, 145, 147, 152, 153. atSymbolNr: 462, 676, 677, 678, 679. $b_{-}GPC: 149.$ AttributeDefinitionObj: 962, 963. baseunix: 89. AttributeDefinitionPtr: 962, 1089, 1143, 1191, begin: 9. BI: 489, 500, 503. AttributeName: 1007, 1009, 1010, 1011, 1019, Biconditional Formula Obj:832, 833. 1066, 1155, 1185. BiconditionalFormulaPtr: 832, 1116, 1478. AttributePatternObj: 937, 938. BinaryArgumentsFormula: 824, 825, 826, 828, AttributePatternPtr: 937, 962, 963, 1066, 1097, 830, 832, 834, 836. 1126, 1143, 1185, 1266, 1311. BinaryFormulaPtr: 824, 1029, 1030, 1031, 1032, AttributeSymbol: 724, 1311, 1395, 1515, 1524, 1033, 1034, 1148, 1163, 1165. 1548, 1608, 1630, 1631, 1641. BinIntFunc: 341, 429, 430, 431, 432, 433, 434, Attribute symbol: 1632.435, 436, 437, 438, 439, 440, 441, 442, 443, AttributiveFormulaObj: 816, 817.<u>444</u>, <u>445</u>, <u>446</u>, 648. AttributiveFormulaPtr: 816, 1038, 1115, 1165, BinIntFuncError: 429, 430, 433, 434, 441, 1467. 443, 444. AttrLookup Table: 1101, 1102, 1104. BinIntFuncPtr: 429. ATTSubexpression: <u>1641</u>, 1643, 1644, 1645. Bl: 1417, 1418, 1420, 1423, 1424, 1425. atUnknown: 462, 1104.blCase: 692, 1187, 1211, 1596. at Value: 462, 676, 679. blDefinition: 692, 1187, 1211, 1634. at VarNr: 462. blDiffuse: 692, 1187, 1211, 1598. at Var Sort: 462. blHereby: 692, 1187, 1211, 1598. atX: 462.blMain: 692, 857, 1016, 1105, 1144, 1202, 1203, atX1: 462.1206. at X2: 462.blNotation: 692, 1187, 1211, 1638. at Y: 462.BlockKind: 692, 693, 694, 699, 855, 856, 858, 859, aTyp: 987, 988, 989, 990, 991, 992, 993, 994, 1101, 1105, 1205, 1206, 1223. 1014, 1021, 1148, 1162. BlockLookup Table: 1101, 1102, 1105. aType: 745, 746, 786, 787, 790, 791, 804, 805, BlockLookUpTable: 1101. 820, 821, 890, 891, 1001, 1002. BlockName: 855, 1016, 1017, 1101, 1144. aTypeChangeList: 900, 901.BlockObj: 693, 694, 695, 696, 697, 698, 699, aTypeExpList: 884, 885, 886, 887, 893, 894, 700, 1205. 895, 896. BlockPtr: 690, 693, 696, 699. aTypeExpr: 900, 901.BlockWrite: 536, 539. aTypeList: 1014, 1058, 1148, 1169.blProof: 692, 1187, 1211, 1213, 1598. at Y1: 462.at Y2: 462.blPublicScheme: 692, 1187, 1211, 1654. blRegistration: 692, 1187, 1211, 1650.AvailableSymbols: 555, 559, 579, 580, 586. a Val: 502, 513, 528, 530.blSuppose: 692, 1187, 1211, 1594, 1596. aValue: 309, 336, 490, 491, 760, 761. BOOL: 101. aVar: 900, 901, 1014, 1022, 1023, 1148, 1157. Boolean: 1108, 1193, 1205, 1227. aVarId: 897, 898, 903, 904. boolean: 34, 35, 53, 61, 62, 63, 64, 103, 105, 107, a Vars: 884, 885, 886, 887. 113, 129, 132, 133, 139, 140, 141, 142, 149, 158, aWSBlock: 1014, 1017, 1148, 1187. 159, 167, 168, 169, 170, 171, 250, 260, 263, 267, aWSItem: 1014, 1079, 1080, 1095, 1148, 1191. 270, 271, 272, 286, 293, 309, 320, 336, 350, 354, aWsItem: 1095.357, 359, 364, 365, 373, 374, 375, 376, 377, 379, aWSTextProper: 1014, 1016, 1096, 1148, 1188, 391, 392, 418, 421, 424, 425, 426, 427, 428, 429, 1192. 437, 440, 447, 455, 458, 487, 492, 528, 530, 558, Axiom: <u>1653</u>, 1660. 593, 595, 604, 628, 631, 632, 643, 644, 751, 879, Axioms: 69. 880, 956, 957, 960, 961, 962, 963, 964, 965, 966, AxiomsAllowed: 64, 69, 1237, 1253, 1363. 967, 1014, 1097, 1138, 1143, 1148, 1165, 1168,

1197, 1205, 1225, 1228, 1232, 1243, 1245, 1292, CircumfixFunctor: 941, 942, 1061, 1064, 1127, 1374, 1388, 1417, 1492, 1500, 1623, 1641. 1129, 1185. borrow: 144, 145. CircumfixFunctorPatternObj: 941, 942. Borrow: 145.CircumfixFunctorPatternPtr: 941, 1064, 1127, Bourbaki, Nicolas: 921, 922. 1185, 1323. Bracket: 766. Circumfix Term Obj: 766, 767.BracketedTerm: 1505, 1511.CircumfixTermPtr: 766, 1048, 1121, 1168, 1443. break: 45, 69, 136, 158, 209, 262, 507, 509, Clear: 309, 317, 402, 410. 619, 622, 633, 1197, 1427, 1635, 1639, 1641, Close: 473, 523, 576, 583, 584. 1650, 1660. close: 46, 110, 111, 472, 479, 486, 497, 596, 599, BuffChar: 520.646, 727, 1009, 1012, 1149. BugInProcessor: 59, 75, 90, 91. CloseEmptyElementTag: 502, 509, 512.CloseErrors: 90, 105, 111. builtin: 128.byte: 46, 48, 49, 53, 59, 74, 75, 185, 464, 477, 478, CloseInfoFile: 486. 498, 562, 563, 589, 653, 655, 657, 1380. CloseInfofile: $\underline{485}$, $\underline{486}$. Byte: 580.CloseParenth: <u>1503</u>, 1518, 1551. *b1*: 137, 143, 144, 147, 152, 153. CloseSourceFile: 723, 731, 733. c: 143, 146, 516. CloseStartTag: 502, 509, 512. Canceled: 1570, 1637, 1650, 1660. Clustered Type Obj: 804, 805. CanceledPragma: 1193, 1194.Clustered Type Ptr: 804, 1021, 1110, 1162, 1411, CapitalizeName: $\underline{1005}$, $\underline{1006}$. 1466. CasePos: <u>1591</u>, 1592, 1594, 1596, 1597. ClusterObj: 987, 988, 989, 990, 991, 992, 993, 994. CatchSignal: 95. ClusterPtr: 987, 1093, 1191. cchBuffer: 45.ClusterRegistrationKind: 987, 988, 1228. CClusterObj: 991, 992.ClusterRegistrationName: 1093, 1143. CClusterPtr: 991, 1093, 1143, 1191, 1272. ClustersProcessing: 61, 65, 66, 67. cdecl: 95, 101. CMinusOne: 131.ch: 604, 628, 631, 632, 644. cmSecs: 119, 120, 121, 123, 124, 126. CH_REPORT: 128, 173, 174, 175, 176, 177. cmt: 632, 642. ChangeFileExt: 34, 43, 48, 49. coConsistentError: 184, 384, 443. char: 30, 32, 45, 185, 485, 516, 533, 538, 558, 562, Code: 309, 321, 341, 342, 402, 412, 520, 521, 563, 583, 585, 589, 591, 592, 596, 597, 601, 524, 525. 602, 604, 628, 631, 632, 644, 665, 666, 673, coDuplicate: <u>184</u>, 314, 385, 441, 444, 451, 460. 1148, 1150, 1193, 1194, 1495. coIndexError: 184, 202, 205, 207, 221, 223, 224, Char: 578, 580, 651, 652, 653, 654, 655, 656. 318, 319, 322, 323, 327, 332, 333, 346, 352, 353, CharKind: 498, 500. 363, 411, 413, 415, 416, 433, 434, 451, 452. CheckCompatibility: $463, \underline{479}$. coIndexExtError: 184, 238, 239, 240, 241, 245, CheckerOnly: 64, 69. 246, 268, 270, 273, 274, 275, 278, 279. CheckTermLimit: 1404, 1405.Col: 85, 103, 104, 107, 108, 109, 485, 495, 498, checkzero: 137, 139, 140, 141, 142, 143, 144, 147, 552, 608, 613, 614, 617, 622, 627, 629, 633, 148, 149, 151, 152, 153, 154, 155, 156, 157. 635, 636, 640, 643, 729, 1017, 1073, 1095, 1100, Choice operator: 790, 1057. 1139, 1144, 1145, 1146, 1498. Choice, Axiom of: 790. CollectBracketForm: 665, 671, 1323. Choice Statement Obj: 906, 907.CollectFormat: 665, 670.ChoiceStatementPtr: 906, 1082, 1143, 1191, 1257. CollectFuncForm: 665, 672, 1323. Choice TermObj: 790, 791. Collective Assumption: 970, 975, 977, 1092, Choice TermPtr: 790, 1057, 1121, 1168, 1451. 1143, 1182. Chr: 32. Collective Assumption Obj: 974, 975, 976.chr: 53, 498, 518, 589, 593, 598, 610, 679. CollectiveAssumptionPtr: 974, 1092, 1143, 1182, CHReport: 173, 174, 175, 176, 177. 1259, 1263. CImUnit: 131. CollectPredForm: 665, 674, 1319.

CollectPrefixForm: 665, 673, 1311, 1314, 1327, ComplexNeg: 134, <u>175</u>, 176. 1329, 1330, 1333. ComplexNorm: 134, 179.CollectToken: 593, 595. ComplexSub: 134, 174. CompactStatementObj: 913, 914, 916.Cond: 105, 113. CompactStatementPtr: 913, 1014, 1075, 1076, Conditional Definiens: 946, 955, 1068, 1186, 1249. 1080, 1097, 1137, 1141, 1148, 1172, 1174, Conditional Definiens Obj: 954, 955. 1191, 1220, 1250, 1261. Conditional Definiers Ptr: 954, 1068, 1130, 1186, Compare: 286, 288, 293, 295, 297, 299, 302, 1385. 304, <u>469</u>, 470. ConditionalFormulaObj: 830, 831. compare: 258, 393. ConditionalFormulaPtr: 830, 1116, 1478, 1488. CompareComplex: 135, 181. Conditional Registration: 987, 992, 1093, 1143, CompareFormats: 647, 659, 660, 661, 662, 663, 1191, 1272, 1281. 664, 675, 676. ConditionalTail: 1539, 1551, <u>1556</u>, 1557. CompareInt: 135, 180, 294, 365, 381, 388, 390, COne: 131, 172, 178. 403, 421, 455. Conjunctive Formula Obj: 826, 827.CompareIntPairs: 294, 304, 354, 437, 461. ConjunctiveFormulaPtr: 826, 1116, 1478. CompareIntStr: 135, 180, 181.Conjunctive Tail: $\underline{1552}$, $\underline{1553}$. CompareNatFunc: 390, 461. Console Mode: 98.CompareProc: <u>250</u>, 251, 257, 259, 267, 268, const: 5. 286, 287. ConstantDefinitionObj: 897, 898. CompareStr: 296, 297, 320, 325, 334, 461. ConstantDefinitionPtr: 897, 1084, 1143, 1191, CompareStringPtr: 282, 283, 461. 1352. Compare With: 379, 393. Construction Type: 1621, 1623, 1624. Complement Of: 359, 369.constructor: 6. CompleteAdjectiveCluster: 716, 718, 1398, 1643, ConstructorsProcessing: 61, 65, 66, 67. 1644, 1645. continue: 613, 617, 624, 626, 1660. CompleteArgument: 1518, 1519, 1520, 1531, 1549. Contradiction: 844. CompleteAtomicFormula: 1544, 1549. ContradictionFormulaObj: 844, 845. CompleteAttributeArguments: 716, 718, 1397, ContradictionFormulaPtr: 844, 1115, 1470. 1524. coOverflow: 184, 345, 352. CompleteAttributes: 717, 718, 1530, 1548, 1641, coOverFlow: 223, 363. 1644, 1645, 1646. Copula: 1297. CompleteAttrIdentify: 707, 708. copy: 136, 137, 150, 534, 582, 614, 619, 622, CompleteClusterTerm: 716, 718, 1399, 1643. 640, 641, 1195, 1196. CompleteClusterType: <u>717</u>, 718, 1643, 1644, Copy: 498, 501, 560, 561, 564, 565, 566, 633, 1645, 1646. 640, 1194. CompleteFuncIdentify: 706, 708, 1283, 1648. $CopyBinIntFunc: \underline{429}, 438.$ CompleteMultiPredicativeFormula: 1540, 1546, CopyCollection: 219, 230. 1547. CopyIntRel: 350, 355. CompletePragmas: 1193, 1197. CopyInt2PairOfIntFunc: 447, 456. CompletePredAntonymByAttr: 707, 708. CopyItems: 195, 218. CompletePredicativeFormula: 1543, 1546, 1547. CopyList: 195, 199, 219, 234, 250, 253. Complete PredIdentify: 707, 708.CopyNatFunc: 379, 380, 389. CompletePredSynonymByAttr: 707, 708. CopyNatSet: 359, 366. Complete Right Side Of The Predicative Formula: 1539,CopyObject: 186, 189, 190, 201, 218, 389. 1540, 1541. Copyright: 71, 78, <u>79</u>. Complete Type: 716, 718, 1411, 1530, 1548, 1643, CopySequence: 402, 405. 1644, 1645, 1646. Corrected: 1417, 1418, 1424. ComplexAdd: 134, 173.Correctness: 1622, 1623, 1643, 1647, 1648. ComplexDiv: 134, <u>177</u>, 178. ComplexInv: 134, $\underline{178}$. Correctness Conditions: 1385. ComplexMult: 134, 176.CorrectnessConditionObj: 980, 981.

CorrectnessConditionPtr: 980, 1079, 1086, 1142, CurIndex: 309, 331. 1143, 1191, 1250. CurlyBracket: 722, 1011, 1321. Correctness Conditions Obj: 982, 983.CurPos: 85, 86, 103, 106, 108, 109, 485, 686, Correctness Conditions Ptr:982, 1087, 1142, 687, 704, 726, 729, 858, 1017, 1079, 1095, 1143, 1250. 1140, 1142, 1143, 1144, 1145, 1146, 1191, 1203. CorrectnessConditionsSet: 982, 983, 1240. 1207, 1211, 1213, 1214, 1215, 1216, 1218, 1219, 1220, 1226, 1229, 1240, 1246, 1280, 1285, 1288, CorrectnessKind: 720, 722, 980, 981, 982, 1080, 1289, 1293, 1294, 1296, 1299, 1310, 1311, 1313, 1101, 1106, 1244. 1318, 1321, 1322, 1325, 1327, 1332, 1333, 1335, Correctness Kind Lookup Table: 1101, 1102, 1106.1339, 1341, 1342, 1347, 1349, 1352, 1353, 1357, CorrectnessName: 722, 1079, 1087, 1101, 1191. 1361, 1362, 1365, 1368, 1369, 1373, 1376, 1378, CorrectnessObj: 978, 979, 980, 982. 1379, 1383, 1392, 1395, 1397, 1399, 1400, 1401, CorrectnessPtr: 978, 1087, 1191. 1403, 1408, 1409, 1410, 1429, 1434, 1435, 1437, coSortedListError: 184, 264, 266, 278, 284, 1438, 1440, 1443, 1445, 1446, 1448, 1450, 1452, 327, 332, 423. 1453, 1454, 1455, 1456, 1457, 1458, 1459, 1460. Count: 195, 197, 198, 199, 201, 202, 203, 204, 205, 1461, 1468, 1470, 1471, 1472, 1473, 1476, 1479, 206, 207, 209, 210, 213, 214, 215, 216, 218, 220, 1480, 1481, 1482, 1483, 1486, 1498, 1524, 1541, 221, 223, 224, 227, 228, 230, 236, 238, 242, 243, 1547, 1577, 1594, 1596, 1597, 1598, 1634, 1638, 244, 245, 246, 247, 249, 251, 252, 253, 254, 255, 1641, 1643, 1649, 1650. 256, 259, 261, 262, 263, 265, 266, 270, 271, 272, Current Year: 77, 79. 273, 276, 277, 278, 279, 284, 289, 293, 306, 341, CurrPatternKind: <u>1630</u>, 1631, 1632. 343, 344, 345, 346, 347, 348, 352, 353, 354, 355, CurrPos: 106.360, 362, 363, 365, 366, 368, 369, 370, 374, 375, 377, 380, 381, 387, 388, 390, 391, 392, 393, 394, CurrWord: 1658.396, 399, 401, 471, 473, 513, 580, 587, 598, 599, CurWord: 723, 726, 1201, 1215, 1217, 1218, 1233, 600, 606, 607, 613, 619, 624, 629, 670, 671, 672, 1242, 1244, 1246, 1283, 1293, 1311, 1313, 1318, 673, 674, 679, 1016, 1017, 1018, 1019, 1020, 1321, 1322, 1323, 1327, 1333, 1337, 1362, 1363, 1021, 1025, 1026, 1035, 1036, 1037, 1043, 1047, 1364, 1365, 1366, 1368, 1369, 1371, 1372, 1375, 1049, 1053, 1054, 1058, 1060, 1068, 1071, 1074, 1376, 1377, 1378, 1379, 1380, 1381, 1392, 1395, 1076, 1077, 1081, 1082, 1083, 1085, 1091, 1092, 1408, 1429, 1440, 1442, 1443, 1444, 1446, 1448, 1094, 1155, 1156, 1159, 1160, 1161, 1162, 1165, 1452, 1454, 1459, 1460, 1461, 1468, 1473, 1496, 1499, 1500, 1503, 1504, 1506, 1507, 1508, 1509, 1168, 1169, 1174, 1176, 1177, 1179, 1181, 1184, 1185, 1186, 1187, 1188, 1191, 1197, 1208, 1263, 1512, 1515, 1518, 1519, 1520, 1521, 1524, 1525, 1526, 1527, 1528, 1531, 1532, 1535, 1538, 1539, 1291, 1411, 1436, 1463, 1466, 1485, 1487, 1488. count: 400.1540, 1541, 1543, 1544, 1545, 1546, 1547, 1548, 1549, 1550, 1552, 1553, 1554, 1555, 1556, 1559, CountAll: 379, 396, 429, 446. 1564, 1570, 1571, 1572, 1573, 1574, 1575, 1576, $create_supposition_block$: <u>1596</u>, 1597. 1577, 1578, 1579, 1580, 1584, 1585, 1586, 1587, create_supposition_head: 1596, 1597. 1588, 1589, 1591, 1592, 1593, 1595, 1596, 1598, CreateArgs: 1199, 1393, 1395, 1409, 1427, 1441, 1599, 1604, 1606, 1607, 1608, 1609, 1612, 1613, 1443, 1445, 1462, 1463, 1469. 1616, 1617, 1621, 1622, 1623, 1624, 1625, 1626, CreateBlock: 693, 699, 1205, 1223, 1594, 1596, 1627, 1628, 1629, 1630, 1633, 1634, 1635, 1636, 1598, 1634, 1638, 1650, 1654. 1637, 1638, 1639, 1640, 1641, 1642, 1643, 1644, CreateExpression: 705, 706, 1351, 1558, 1561, 1647, 1648, 1649, 1650, 1654, 1655, 1656, 1657, 1643, 1644, 1645, 1646, 1651. 1658, 1659, 1660, 1661, 1662. CreateItem: 693, 698, 1205, 1222, 1569, 1570, Curword: 1511, 1514, 1591, 1595, 1611. 1580, 1581, 1582, 1583, 1584, 1587, 1588, 1589, CZero: 131, 177. 1591, 1594, 1596, 1597, 1615, 1622, 1623, 1624, 1631, 1632, 1633, 1634, 1637, 1638, 1640, 1643, dArgList: 1426, 1427.DARWIN: 81. 1647, 1648, 1649, 1650, 1651, 1652, 1653, 1654. CreateSubexpression: 710, 712, 1489, 1491, 1519, Data: 187. 1530, 1531, 1549, 1561, 1643, 1644, 1645, 1646. Dct: <u>596</u>, 597. CtrlCPressed: 64, 73, 95, 99. .dct file: 596, 599, 600, 1009, 1010. CtrlSignal: 97, 98, 99, 100, 101. DctFile: 599.

.dcx file: 596. Dereferencing: 194. DEBUG: 128. $destructor: \underline{6}.$ DEBUGNUM: 128, 137, 139, 140, 141, 142, 143, dfEmpty: 946, 1143, 1191, 1229, 1249, 1253. 144, 146, 147, 148, 149, 150, 151, 152, 153, dfEquals: 946, 1143, 1191, 1299, 1300, 1304, 1385. 154, 155, 156, 157. dfMeans: 946, 1143, 1191, 1299, 1303, 1304, 1385. Dec: 221, 257, 279, 318, 334, 346, 403, 411, 434. dicthan: 554, 647. dec: 214, 246, 257, 272, 384, 443, 452, 546, 548, DiffuseStatementObj: 911, 912.616, 639, 1005, 1146, 1213, 1221, 1406, 1427, DiffuseStatementPtr: 911, 1076, 1141, 1174, 1261. 1466, 1467, 1503, 1521, 1522, 1532, 1533, Dir: 39, 40, 41, 42. 1545, 1551, 1577. DirectiveKind: 720, 722. DefaultAllowed: 590, 645. DirectiveName: 722.DefaultExt: 48, 49. DirSeparator: 77, 474, 482, 483. defExpandableMode: 956, 959, 1088, 1143, 1191. disable_io_checking: 11, 71, 86. DefExpressionObj: 948, 949.DisjunctiveFormulaObj: 828, 829. DefExpressionPtr: 948, 950, 951, 952, 953, 954, DisjunctiveFormulaPtr: 828, 1116, 1478. 955, 1130, 1304, 1385. Disjunctive Tail: 1554, 1555. DisplayLine: 58, 71, 85, 86, 686, 687, 704, 1079, DEFINED: 10.DefiniensKindName: 1068, 1130. 1144, 1145, 1146, 1191. DefiniensObj: 946, 947, 950, 954. DisplayLineInCurPos: 59, 71, 73. DefiniensPtr: 946, 960, 961, 962, 963, 964, Dispose: 188, 211, 245, 249, 266, 278, 291, 965, 966, 967, 1014, 1068, 1097, 1130, 1148, 311, 994, 996. 1186, 1228. dispose: 523, 585, 595, 624, 676, 684, 685, 686, DefiniensSort: 946, 947. 687, 731, 744, 746, 750, 752, 763, 765, 767, 769, Defining WayName: 1079, 1143. 771, 773, 777, 779, 783, 785, 787, 789, 791, 799, Definition: <u>1623</u>, 1635. 805, 811, 813, 815, 817, 819, 821, 823, 825, 839, DefinitionalBlock: 1634, 1660. 859, 861, 865, 878, 885, 887, 889, 891, 894, 896, DefinitionReference: 867, 872, 1071, 1134, 1176. 898, 901, 904, 907, 912, 914, 917, 918, 927, 931, 936, 938, 940, 942, 947, 949, 951, 953, 955, DefinitionReferenceObj: 871, 872. $Definition Reference Ptr \colon \quad 871, \ 1071, \ 1134, \ 1176,$ 957, 959, 961, 963, 965, 967, 969, 973, 975, 1376. 977, 979, 988, 990, 992, 994, 996, 998, 1002, DefinitionsProcessing: 61, 65, 66. 1004, 1012, 1096, 1147, 1192, 1263. DefPatternName: 1062, 1065, 1066, 1091, 1101. Dispose Formats: 680.DefPos: 1634, 1638, 1650. DisposePrf: 723, 728, 733. defStandardMode: 956, 961, 1088, 1143, 1191. DisposeStr: 298, 311, 317, 318, 339, 461, 468. delete: 138, 479, 637, 639, 641, 1194. $Dispose WS Lookup Tables \colon \ \underline{1102}, \ 1147.$ Delete: 26, 28, 219, 225, 226, 309, 318, 331, 475, Div: 129, 149, 157.476, 477, 478, 518, 558. div: 7. $delete_prefix$: 558. diva: 160.DeleteAll: 195, 198, 210, 216, 217, 235, 239, Diva: 129, 153. 252, 341, 348, 367, 380, 429, 436, 447, 454, DivA: 157.1291, 1411, 1436. Divides: 129, 159. DeleteElem: 359, 369, 372. div1: 148, 149. DeleteExtItems: 235, 248. $Div1: \underline{148}.$ DeleteFile: 35.Do-support: 1540. $DeleteInt: \underline{418}, \underline{420}.$ do: 1540. DELPHI: 25, 38, 44, 81, 96, 101, 119. does: 1540. Delta: 219, 220, 223, 229, 230, 233, 234, 324, 359, dol_Allowed: 1227, 1229, 1348, 1349, 1354, 1454. 360, 361, 363, 366, 367, 380, 381. Done: 18, 186, 188, 195, 200, 211, 216, 219, 220, Den: 130, 131, 160, 161, 162, 163, 164, 165, 167, 235, 237, 245, 249, 266, 267, 269, 278, 291, 307, 168, 169, 170, 181. 308, 309, 311, 341, 344, 359, 362, 386, 402, 407, Depo: 1421, 1426, 1427. <u>429</u>, 432, <u>447</u>, 450, <u>466</u>, 468, <u>494</u>, 497, <u>502</u>, 504, DepoNbr: 1426, 1427.<u>520</u>, 523, <u>524</u>, 527, <u>533</u>, 536, 537, <u>562</u>, 570, <u>571</u>,

572, 573, 575, 585, 595, 600, 604, 609, 624, 632, *eEnd*: 489, 506, 508, 510, 676, 1107, 1109, 1117, 646, 676, 679, 680, 684, 685, 686, 687, 693, 696, 1122, 1124, 1130, 1134, 1143, 1146. 703, 704, 713, 715, 728, 731, 743, 744, 745, 746, EI: 489, 500, 503. <u>747</u>, 748, <u>749</u>, 750, <u>751</u>, 752, <u>762</u>, 763, <u>764</u>, 765, elAdjective: 462, 1019.<u>766</u>, 767, <u>768</u>, 769, <u>770</u>, 771, <u>772</u>, 773, <u>776</u>, 777, elAdjective Cluster: 462, 1020. <u>778</u>, 779, <u>782</u>, 783, <u>784</u>, 785, <u>786</u>, 787, <u>788</u>, 789, elAncestors: 462, 1091.790, 791, 798, 799, 800, 801, 802, 803, 804, 805, ElapsedTime: 116, 123, 126. 810, 811, 812, 813, 814, 815, 816, 817, 818, 819, elArguments: 462, 1035, 1036, 1047.820, 821, 822, 823, 824, 825, 838, 839, 856, 857, elArticleID: 462. 859, 860, 861, 864, 865, 877, 878, 879, 880, 881, elBlock: 462, 1017, 1146. 882, 884, 885, 886, 887, 888, 889, 890, 891, 893, elConditions: 462, 1082, 1085, 1092. 894, 895, 896, 897, 898, 900, 901, 903, 904, 906, elCorrectnessConditions: 462, 1087. $907,\,\underline{909},\,910,\,\underline{911},\,912,\,\underline{913},\,914,\,\underline{915},\,\underline{916},\,917,$ elDefiniens: 462, 1068, 1130. 918, 926, 927, 930, 931, 935, 936, 937, 938, elDirective: 462.939, 940, 941, 942, 946, 947, 948, 949, 950, $ElemLookup Table: \quad \underline{1101}, \ 1102, \ 1103.$ 951, <u>952</u>, <u>953</u>, <u>954</u>, <u>955</u>, <u>956</u>, <u>957</u>, <u>958</u>, <u>959</u>, ElemNr: 359, 378.960, 961, 962, 963, 964, 965, 966, 967, 968, elEnviron: 462.969, <u>972</u>, 973, <u>974</u>, 975, <u>976</u>, 977, <u>978</u>, 979, elEquality: 462, 1083, 1143. 980, 981, 982, 983, 984, 985, 987, 988, 989, elFieldSegment: 462, 1091, 1143. 990, <u>991</u>, 992, <u>993</u>, 994, <u>995</u>, 996, <u>997</u>, 998, elFormat: 462, 676, 678. 999, 1000, 1001, 1002, 1003, 1004, 1012, 1014, elFormats: 462, 676, 679. 1015, 1096, <u>1097</u>, 1098, 1102, 1147, <u>1148</u>, 1149, elIdent: 462.1192, 1263, 1291, 1436, 1485, 1487. elItem: 462, 1095, 1144, 1145. dos: 25, 120. elIterativeStep: 462, 1076, 1141. Down: 379, 384, 429, 443. elLabel: 462, 1067, 1131. DrawArticleName: 57, 72, 87. elLink: 462, 1072, 1138. DrawErrorsMSq: 76. elLoci: 462, 1060, 1143. DrawErrorsMsg: 59, 88. elLociEquality: 462, 1094, 1143. DrawIOResult: 46, 59, 75. elLocus: 462, 1059.DrawMessage: 46, 47, 50, 59, 74, 75, 90, 91, elNegatedAdjective: 462, 1019. elPartialDefiniens: 462, 1068, 1130. 106, 108, 481. elPriority: 462, 676, 679. DrawMizarScreen: 57, 71, 87. elProposition: 462, 1069, 1143. DrawPass: 59, 72.elProvisionalFormulas: 462, 1081, 1143. DrawStr: 57, 71. elRedefine: 462, 1088, 1089, 1090, 1143. DrawTime: 59, 72, 85. elRightCircumflexSymbol: 462, 1048, 1064. DrawTPass: 71. elSchematic Variables: 462, 1081. DrawVerifierExit: 59, 72, 86. elScheme: 462, 1081. DT: 489, 500. else: 8, 10. dupAccept: 309, 320.ELSE: 10, 77, 80, 92. dupError: 309, 310, 314. $else_def: 10.$ dupIqnore: 309, 314. $else_if_def: 10.$ Duplicates: 286, 287, 289, 290, 291, 293, 359, ELSEIF: 10.360, 363, 365. elSelector: 462, 1091, 1143. DWORD: 98, 100, 101. elSetMember: 462.Dynamic Array: 195. elSkippedProof: 462. e: 1101.elSubstitution: 462.EBNF: 855. elSymbol: 462, 587, 600, 1012. ec: 532.elSymbolCount: 462, 586.EClusterObj: 989, 990.elSymbols: 462, 587, 600. EClusterPtr: 989, 1093, 1143, 1191, 1272. elTypeList: 462, 1058. $elType Specification {:}\quad 462,\, 1081,\, 1088,\, 1090,\, 1143.$ EE: 489, 500, 506, 507, 508, 509.

elUnknown: 462, 1103. 1459, 1496, 1501, 1573, 1575, 1596, 1631, 1632, 1633, 1643, 1644, 1645, 1647, 1655, 1658, 1662. elVariable: 462, 1022, 1113, 1135, 1143. errMissingXMLAttribute: 488, 531. elVariables: 462, 1025, 1077, 1081. elVocabularies: 462, 585. ErrMsg: 60, 75, 90. elVocabulary: 462, 585.ErrNr: 85, 103, 105, 106, 107. EmptyNatFunc: 461. Error: 105, 106, 520, 521, 524, 525, 1253, 1276, 1363, 1395, 1409, 1419, 1420, 1423, 1424, 1425, EmptyParameterList: 59, 71, 87. EN: 489.1445, 1447, 1449, 1462, 1463, 1496, 1499, 1636. Error, 58: 1596. $enable_io_checking: 11, 86.$ Error, 65: 1458. ENABLE_PROCESSED_INPUT: 98. Error, 73: end: 8, 9, 10. 1363.Error, 146: 1376, 1377. end_empty_tag : 509. Error, 151: end_if: 10. 1409.Error, 152: 1443. end_mdebug: $\underline{9}$. $end_start_tag: \underline{509}.$ Error, 153: 1462. Error, 157: endcases: 8. 1459. Error, 175: 1394. ENDIF: 9, 10, 77, 80, 89, 92, 95, 96, 100, 101, Error, 176: 119, 120, 128, 137, 139, 140, 141, 142, 143, 144, 1445. 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, Error, 178: 1276.156, 157, 173, 174, 175, 176, 177, 187, 463. Error, 181: 1454. Error, 182: endif: 10. 1447.EndOfIdent: 612, 613, 616, 617, 622, 624, 626. Error, 184: 1449. EndOfPhrase: 618, 619, 620. Error, 185: 1409. EndOfSymbol: 612, 613, 619, 620. Error, 200: 617.Error, 201: EndOfText: 604, 628, 629, 632, 636, 643. 617.EnlargeBy: 359, 368, 379, 387, 388, 389. Error, 202: 625.entry_is_uninitialized: 1606, 1613, 1618. Error, 203: 627. $entry_is_unitialized$: 1606. Error, 231: 1593.EnvFileExam: 47. Error, 232: 1594.Error, 251: EnvFileName: 24, 47, 50, 52, 482, 733, 1208. 1453. EnvMizFiles: 464, 484. Error, 300: 1566, 1602. Error, 306: eof: 472, 499, 582, 583, 643. 1608. EOT: 603, 614, 629, 724, 1497, 1577, 1587, 1595, Error, 307: 1575.1635, 1639, 1650, 1658, 1659, 1660. Error, 308: 1571, 1574. Error, 313: EOTX: 489, 499, 503, 506, 507, 508, 509. 1575.EQ: 489, 500, 507, 509. Error, 314: 1632.EqualitiesProcessing: 61, 65, 66. Error, 336: 1556.Equality Sym: 722, 1011, 1613.Error, 350: 1567, 1568. Equals: 364, 374, 375, 387, 388. Error, 351: 1593. Equating: 900, 1083, 1143, 1191, 1357. Error, 370: 1574, 1607, 1608. Erase: 537. Error, 401: 1540, 1555. *erase*: 110. Error, 402: 1553. EraseErrors: 105, 110.Error, 403: 1567.EraseFile: 34, 35, 36, 473, 533, 537. ErrorAddr: 90. Err: 489, 500. ErrorCode: 90, 105, 112, 113, 114. errBadXMLToken: 488, 505. ErrorNbr: 85, 86, 88, 90, 103, 106, 108, 109, 686, errElMissing: 488, 510. 687, 704, 1079, 1191. errElRedundant: 488, 510. ErrorRecovery: 502, 503, 505, 506, 507, 508, 509. errhan: 83, 89, 102, 485, 487, 514, 588, 647, 681, ErrorReport: 103.719, 735, 850, 1199, 1492. Errors: 107, 108, 109, 110, 111. ErrImm: 90, 105, 106, 726, 1218, 1233, 1237, 1242, ErrorSymbol: 603, 617, 626, 627. 1244, 1307, 1376, 1377, 1399, 1453, 1454, 1458, errWrongXMLElement: 488, 492.

eStart: 489, 506, 509, 510, 1012, 1113, 1121, extBlockObj: 1205, 1206, 1213, 1214, 1215, 1216, 1127, 1130, 1131, 1135, 1140, 1141, 1143, 1217, 1218, 1219, 1221, 1222, 1223. 1144, 1145, 1146. extBlockObj.ProcessBegin: 1214.extBlockPtr: 1203, 1205, 1211, 1223, 1276, 1362, ET: 489, 500, 506, 507, 508, 509. ex: 152, 154, 155, 156, 157. 1379, 1380, 1381. ExactlyTermObj: 786, 788, 789. ExtBlockPtr: 1230, 1244, 1295. ExactlyTermPtr: 788, 1056, 1121, 1168, 1403. exTerm: 709, 1068, 1130, 1186, 1304, 1385, exAdjectiveCluster: 18, 709, 1641, 1643, 1644, 1558, 1642, 1643. 1645. $extExpressionObj: 1489, \underline{1490}, \underline{1491}.$ ExampleObj: 903, 904.extExpressionPtr: 1351, 1489.ExamplePtr: 903, 1086, 1143, 1191, 1361, 1362. extItemObj: 1225, 1226, 1249, 1277, 1279, 1280, Exchange: 309, 319, <u>1417</u>, 1418, 1423, 1424. 1281, 1282, 1283, 1285, 1286, 1288, 1289, ExchangeItems: 309, 319, 334. 1290, 1291, 1293, 1294, 1295, 1296, 1299, 1300, exclude: 1244.1301, 1303, 1304, 1305, 1306, 1307, 1310, 1311, Exercises: 851. 1312, 1313, 1314, 1315, 1318, 1319, 1321, 1322, exFormula: 709, 1068, 1130, 1186, 1304, 1385, 1323, 1324, 1325, 1326, 1327, 1329, 1330, 1332, 1558, 1561. 1333, 1334, 1335, 1337, 1339, 1341, 1342, 1343, Existential Assumption: 970, 1569, 1587, 1637. 1344, 1345, 1347, 1349, 1350, 1351, 1352, 1353, Existential Assumption Obj: 976, 977. 1354, 1355, 1356, 1357, 1358, 1359, 1360, 1361, 1362, 1363, 1364, 1365, 1366, 1368, 1369, 1371, ExistentialAssumptionPtr: 976, 1085, 1143, 1372, 1373, 1375, 1376, 1377, 1378, 1379, 1380, 1191, 1257. 1381, 1383, 1384, 1385. Existential Formula: 1537, 1549. ExistentialFormulaObj: 842, 843. extItemPtr: 1222, 1225.ExistentialFormulaPtr: 842, 1115, 1487. Extract: 562, 564, 568, 577. ExistentialRegistration: 987, 990, 1093, 1143, $ExtractFileDir: 34, \underline{41}, 482.$ 1191, 1229, 1272. ExtractFileExt: 34, 38, 42, 48, 49, 108, 109, 581. exit: 48, 49, 50, 53, 139, 148, 150, 181, 205, 221, $ExtractFileName: 34, 38, \underline{42}, 50.$ 223, 238, 239, 240, 241, 245, 246, 255, 262, 264, ExtractFilePath: 38. 266, 278, 284, 302, 306, 346, 352, 358, 363, 374, extSubexpObj: <u>1388</u>, 1390, 1391, 1392, 1395, 1396, 375, 377, 382, 387, 388, 390, 391, 392, 393, 401, 1397, 1398, 1399, 1400, 1401, 1402, 1403, 1405, 414, 419, 425, 426, 428, 433, 434, 441, 451, 452, 1406, 1407, 1408, 1409, 1410, 1411, 1413, 1417, 474, 475, 476, 477, 478, 499, 530, 531, 558, 559, 1429, 1430, 1431, 1432, 1433, 1434, 1435, 1436, 561, 568, 576, 581, 582, 606, 607, 614, 629, 636, 1437, 1438, 1439, 1440, 1441, 1442, 1443, 1444, 640, 1020, 1376, 1377, 1577, 1587, 1590, 1595. 1445, 1446, 1447, 1448, 1449, 1450, 1451, 1452, Exit: 314, 325, 329. 1453, 1454, 1455, 1456, 1457, 1458, 1459, 1460, ExitCode: 92.1461, 1462, 1463, 1464, 1465, 1466, 1467, 1468, ExitProc: 86, 87, 89, 90, 92, 93, 94, 486. 1469, 1470, 1471, 1472, 1473, 1474, 1475, 1476, exNull: 709, 1641, 1642, 1643. 1477, 1478, 1479, 1480, 1481, 1482, 1483, 1484, ExpandableModeDefinitionObj: 958, 959. 1485, 1486, 1487, 1488. ExpandableModeDefinitionPtr: 958, 1088, 1143, extSubexpPtr: 1388, 1491.1191, 1264. exType: 709, 1558, 1643, 1644, 1645, 1646. ExpansionsProcessing: 61, 65, 66. False: 260, 270, 287, 293, 320, 354, 360, 365, ExpKind: 705, 706, 709, 710, 711, 948, 949, 1068, 421, 437, 455. 1130, 1351, 1489, 1490, 1641, 1643, 1647. false: 53, 62, 64, 65, 66, 67, 70, 85, 103, 107, 110, ExplicitlyQualifiedSegmentObj: 745, 746. 111, 113, 139, 140, 158, 169, 263, 271, 272, 310, Explicitly Qualified Segment Ptr: 745, 1025, 1113, 374, 375, 377, 379, 386, 391, 392, 421, 425, 426, 1159, 1168, 1291, 1436, 1485. 428, 505, 530, 531, 558, 595, 619, 628, 959, ExpName: 1068, 1130. 1015, 1098, 1138, 1143, 1149, 1168, 1195, 1197, 1207, 1211, 1214, 1215, 1229, 1242, 1276, 1303, ExpressionObj: 710, 711, 712, 1489. ExpressionPtr: 690, 705, 710. 1380, 1381, 1422, 1459, 1496, 1500, 1623, 1624. exResType: 709, 1651. fAnother: 195, 216, 217, 219, 229, 359, 368, Ext: 39, 40, 41, 42. 369, 370, 379, 388.

File, .idx: 733, 1012. fArgsNbr: 657, 658, 663, 678, 1505, 1506, 1533, File, .prf: 727. 1534. fAt: 533, 549, 550, 553. File, .wsx: 850. fBlockKind: 693, 694, 699, 1205, 1206, 1223. FileAge: 37.fBuffCount: 520, 522, 536, 539. FileDescr: 466, 467, 468. fBuffInd: 520, 522, 536, 538, 539. File Descr Collection: $465, 469, \underline{470}, \underline{471}, \underline{472}, \underline{473}$. fCapacity: 309, 310, 324, 328, 335, 402, 404, 405, FileExam: 46, 87, 475, 476, 477, 478, 480, 406, 408, 415, 417, 419, 429, 431, 433, 435, 583, 1208. 438, 447, 449, 451, 453, 456. FileExists: 35. *fCh*: 1500. File Mode:46. FClusterObj: 993, 994. FileName: 59, 75, 105, 108, 109, 477, 478. FClusterPtr: 993, 1093, 1143, 1191, 1272. FillChar: 187, 572, 574. fCompare: 250, 251, 252, 253, 254, 255, 259, Find: 18, 250, 255, 260, 261, 262, 263, 264, 267, 260, 261, 264, 265, 267, 268, 270, 271, 272, 270, 271, 272, 284, 309, 313, 320, 325, 418, 419, 283, 284, 286, 287, 288. 420, 421, 422, 424, 666, 667, 668, 669, 670. fCount: 309, 310, 311, 313, 316, 317, 318, 319, fIndex: 250, 251, 252, 253, 254, 255, 256, 259, 320, 322, 323, 325, 327, 328, 329, 332, 333, 261, 263, 264, 266, 267, 270, 271, 272, 273, 277, 335, 337, 402, 404, 406, 407, 408, 409, 410, 278, 279, 284, 598, 618, 666, 667, 668, 669. 411, 413, 414, 415, 416, 417, 419, 421, 425, FindInterval: 18, 267, 272. 426, 428, 429, 431, 432, 433, 434, 435, 436, FindRight: 267, 271, 279. 437, 438, 445, 446, 447, 449, 450, 451, 452, fInFile: 647, 677. 453, 454, 455, 456, 584, 585, 679. fInfinitive: 18, 562, 563. fCurrentLine: 632, 633, 635, 636, 637, 638, 639, FinishAdjectiveCluster: 717, 718, 1523. 640, 641, 643, 645, 646. FinishAggregateTerm: <u>716</u>, 718, 1445, 1510. fDelta: 219, 231. FinishAggrPattSegment: 706, 708, 1334, 1620.fDuplicate: 309, 310, 314, 320. FinishAntecedent: 706, 708, 1281, 1645. FinishArgList: 716, 718, 1430, 1519. fEl: 533, 543, 545, 547, 548, 551. fElem: 359, 372, 373, 379, 385, 397, 400. FinishArgument: 716, 718, 1405, 1427, 1459, 1514, $1518,\ 1519,\ 1520,\ 1531,\ 1533,\ 1534.$ fErrNbr: 57, 58, 59, 71, 73. fErrNr: 1492, 1496, 1498, 1500, 1501. FinishAssumption: 707, 708, 1295, 1564, 1568. fExpKind: 705, 706, 710, 711, 1351, 1489, 1490. FinishAtSignProof: 693, 697, 1577. fExt: 464, 474. FinishAttributeArguments: 716, 718, 1397, 1642. fExtCount: 235, 236, 238, 239, 240, 241, 242, FinishAttributePattern: <u>706</u>, 708, 1311, 1608. 243, 245, 246, 247, 248, 249, 274, 276, 277, FinishAttributes: 717, 718. 278, 279, 280. FinishAttributiveFormula: 716, 718, 1467, 1548. fExtCounter: 238. FinishBinaryFormula: 716, 718, 1478, 1552, FF: 1606, 1617, 1618. 1554. 1556. fFileBuff: 520, 522, 523, 536, 538, 539. FinishBracketedTerm: 716, 718, 1443, 1505, 1514. fFirstChar: 593, 598, 619. FinishChoice: 707, 708, 1567. fFunc: 359, 366, 367, 374, 375, 376, 377, 379, FinishChoiceTerm: 716, 718, 1451, 1515, 1516, 380, 387. 1517. fIdent: 466, 467. FinishClusterTerm: 706, 708, 1282, 1644. fIdents: 604, 606, 608, 609, 624, 733. FinishClusterType: 707, 708, 1643, 1644, 1645, FieldSegmentObj: 926, 927.FieldSegmentPtr: 926, 1091, 1143, 1191, 1334. FinishCompactStatement: 706, 708, 1381, 1599.FieldSymbolObj: 924, 925. FinishCondition: 706, 708, 1359, 1567. FieldSymbolPtr: 924, 1091, 1143, 1191, 1333. FinishConsequent: 706, 708, 1281, 1643, 1644, FIL: 472, 473. 1645, 1646. FinishConstructionType: 706, 708, 1306, 1621. File: 520.File, .fil: FinishDefaultTerm: 706, 708, 1358, 1584. 472.File, .dct: 1009, 1010. FinishDefiniens: 706, 708, 1385, 1626, 1628. File, .frt: FinishDefinition: 693, 697, 1623. 850.

FinishDrawing: 57, 71, 88. FinishRightSideOfPredicativeFormula: 716, 718, FinishEquality: 717, 718. 1463, 1539. $Finish Sample: \quad \underline{717}, \ 718, \ 1513.$ FinishExemplifyingTerm: <u>706</u>, 708, 1362, 1588. FinishExemplifyingVariable: 706, 708, 1361, 1588. FinishScanning: 723, 733. FinishSchemeDeclaration: 707, 708, 1654. FinishExistential: 716, 718, 1487, 1537. FinishFields: 706, 708, 1330, 1619. FinishSchemeDemonstration: 693, 700, 1205, FinishFixedSegment: 706, 708, 1291, 1566. <u>1221</u>, 1657. FinishFixed Variables: 706, 708, 1293, 1565. FinishSchemeHeading: 707, 708, 1654. FinishFlexConjunction: 716, 718, 1480, 1553. FinishSchemePremise: 706, 708, 1345, 1656. $FinishScheme Qualification: \underline{706}, 708, 1339, 1655.$ FinishFlexDisjunction: <u>716</u>, 718, 1479, 1555. FinishSchemeReference: 707, 708, 1574. FinishForgetfulTerm: 716, 718, 1449, 1516. FinishFraenkelTerm: <u>716</u>, 718, 1437, 1513. FinishSchemeSegment: 706, 708, 1343, 1655. FinishFuncReduction: 707, 708, 1647. FinishSchemeThesis: 706, 708, 1344, 1654. FinishFunctorPattern: 706, 708, 1323, 1609, FinishSchLibraryReferences: 707, 708, 1575. FinishSelectorTerm: <u>716</u>, 718, 1447, 1515. 1610, 1611. FinishGuard: 706, 708, 1304, 1627, 1629. FinishSentence: 707, 708, 1560, 1561. FinishHypothesis: 706, 708, 1360, 1563, 1564. FinishSethoodProperties: 706, 708, 1312, 1649. FinishingPass: 64, 73, 85. FinishSimpleFraenkelTerm: 716, 718, 1439, 1517. $FinishIterativeStep: \underline{706}, 708, \underline{1384}, 1601.$ $FinishSimpleJustification: \underline{706}, 708, \underline{1380}, 1576.$ FinishJustification: 707, 708, 1578, 1599, 1600, FinishSpecification: <u>706</u>, 708, 1305, 1614. 1601. FinishTerm: 716, 718, 1406, 1531. FinishLocus Type: 706, 708, 1350, 1579. FinishTheLibraryReferences: 707, 708, 1573. FinishLongTerm: 716, 718, 1417, 1519. FinishTheorem: 707, 708, 1570, 1652, 1653. FinishModePattern: 706, 708, 1314, 1604. Finish Theorem Body: 707, 708, 1652, 1653. FinishMultiPredicativeFormula: 716, 718, 1465, Finish Type: 716, 718, 1409, 1525. 1540. FinishUniversal: 716, 718, 1488, 1538. FinishOtherwise: 706, 708, 1300, 1626, 1627, Finish Visible: 707, 708, 1603, 1607. 1628, 1629. Finitary metatheory: 920, 922. FinishPass: 84, 85. fInt: 299, 300. FinishPostqualifyingSegment: 716, 718, 1507. FirstThat: 299, 306. FinishPostQualifyingSegment: 1436.fItemKind: 693, 698, 703, 704, 1205, 1222. FinishPredicatePattern: 706, 708, 1319, 1612. Fixed Variables: <u>1565</u>, 1567, 1568, 1640, 1650. FinishPredicativeFormula: 716, 718, 1462, 1543. fKey: 299, 303, 304, 306. FinishPrefix: 706, 708, 1326, 1616. fKind: 562, 563, 1225, 1226. FinishPrivateConstant: 706, 708, 1352, 1583. fLatAtom: 379, 389. FinishPrivateFormula: <u>716</u>, 718, 1469, 1550. fLeftArgsNbr: 655, 656, 658, 664, 678. FinishPrivateFuncDefinienition: 706, 708, 1355, FlexaryConjunctiveFormulaObj: 834, 835. 1581. FlexaryConjunctiveFormulaPtr: 834, 1115, 1480. FinishPrivatePredDefinienition: 706, 708, 1356, FlexaryDisjunctiveFormulaObj: 836, 837. 1582. FlexaryDisjunctiveFormulaPtr: 836, 1115, 1479. FinishPrivateTerm: 716, 718, 1441, 1509. FlexConjunctiveTail: 1553, 1554. FinishQualifiedSegment: <u>716</u>, 718, 1485, 1535. FlexDisjunctive Tail: 1555, 1556.FinishQualifiedTerm: <u>716</u>, 718, 1402, 1504. fLexem: 591, 592, 599, 600, 602, 606, 607, 608, FinishQualifyingFormula: 716, 718, 1466, 1548. 613, 624, 629, 630, 726, 729. FinishQuantified: 717, 718, 1537, 1538. fLine: 57, <u>58</u>, 59, 71, 73, 632, 642. FinishReconsideredTerm: 706, 708, 1357, 1584.fList: 309, 310, 311, 316, 317, 318, 319, 320, 322, FinishReconsidering: 707, 708, 1584. 323, 325, 326, 328, 332, 333, 334, 335, 402, FinishReferences: 707, 708, 1571. 404, 406, 408, 409, 411, 413, 414, 415, 416, FinishReservation: 707, 708, 1651. 417, 419, 421, 425, 426, 428, 429, 431, 433, FinishReservationSegment: 706, 708, 1347, 1651. 434, 435, 437, 438, 442, 443, 444, 445, 446, FinishRestriction: 716, 718, 1477, 1538. 447, 449, 451, 452, 453, 455, 456, 460, 464,

<u>475</u>, <u>476</u>, <u>477</u>, <u>478</u>, 584, 585, 679, 1526, 1527, FreeItem: 195, 211, 213, 222, 226, 295, 298. 1539, 1543, 1606, 1613, 1618. FreeItemsFrom: 195, 212, 213, 235, 245, 250, fName: 57, 72, 464, 469, 472, 473, 474, 475, 476, 266, 267, 278. 557, 581, 582, 665, 676, 679. FreeMem: 215, 243, 256, 277, 335, 339, 347, 388, fNatFunc: 379, 391, 392, 393. 417, 435, 497, 527, 646. fObject: 309, 311, 316, 323, 326, 328, 333, 584, 585. fRepr: 562, 563.ForgetfulFunctor: 724, 1630. fRightArgsNbr: 653, 654, 656, 658, 662, 664, 678. ForgetfulFunctorTermObj: 778, 779. fRightSymbolNr: 657, 658, 663, 678. $Forgetful Functor TermPtr: \quad 778, \ 1051, \ 1121, \ 1168,$.frt file: 850. 1449. fSing: 219, 231. $fOrigin: \underline{195}, \underline{218}.$ fSorted: 309, 310, 313, 325, 327, 332, 336, 337, 583. formats: 647, 1199. fSourceBuff: 632, 645, 646. FormatsProcessing: 61, 65, 66, 67. fSourceBuffSize: 632, 645, 646. FormulaExpression: <u>1558</u>, 1560, 1582, 1626, fSourceFile: 632, 636, 643, 645, 646. 1627, 1629, 1654. fStart: 632, 642. FormulaExpressionObj: 808, 810, 814, 816, 818, fStr: 191, 192, 282, 592, 598, 599, 600, 602, 606, 820, 822, 824, 838, 844, 846, 848. 607, 613, 619, 620, 629, 630, 726, 729. fString: 309, 311, 316, 317, 318, 320, 322, 325, FormulaKindLookupTable: 1101, 1102, 1105. FormulaName: 1026, 1027, 1028, 1029, 1030, 328, 332, 334, 584, 585. 1031, 1032, 1033, 1034, 1035, 1036, 1037, 1038, fSymbol: 651, 652, 654, 656, 658, 659, 660, 1039, 1040, 1041, 1101. 661, 678. FormulaPtr: 784, 785, 808, 822, 823, 824, 825, fSymbolCnt: 571, 572, 574, 577, 579, 580, 586. 826, 827, 828, 829, 830, 831, 832, 833, 834, fTime: 121, 466, 467. 835, 836, 837, 838, 839, 840, 841, 842, 843, fTokens: 604, 607, 608, 609, 619, 645. 864, 865, 888, 889, 895, 896, 952, 953, 1014, fTokensBuf: 604, 608, 609, 613, 614, 617, 624, 1027, 1068, 1097, 1115, 1148, 1165, 1186, 1205, 626, 627, 629, 630. 1212, 1248, 1260, 1388, 1463. func_type: 1387, 1388. FormulaSort: 808, 1101, 1105. FuncInstNr: 1426, 1427.FormulaSubexpression: 1506, 1513, 1537, 1538, FuncPos: 1387, 1417, 1419, 1420, 1423, 1424, 1557, 1558. 1425, 1427, 1429. forward: 1504, 1505, 1506, 1539, 1577, 1580. function: 6. FoundToken: 612, 613, 618, 619, 620. FunctorDefinitionObj: 966, 967. fOutFile: 651, 678.FunctorDefinitionPtr: 966, 1079, 1090, 1143, fParam: 464, 477, 478. 1191, 1265. fParenthCnt: 1503, 1518, 1519.FunctorialRegistration: 987, 994, 1093, 1143, fPassName: 71. 1191, 1272, 1282. FPC: 25, 38, 44, 81, 89, 95, 96, 100, 120. FunctorName: 1007, 1009, 1010, 1047, 1063, fPhrase: 604, 608, 609, 613, 614, 616, 617, 619, 1168, 1185. 622, 626, 627, 633, 640. FunctorPatternName: 1063, 1064, 1127, 1129. fPhrasePos: 604, 608, 613, 617, 622, 627, 629, FunctorPatternObj: 941, 942. 633, 635, 636, 640, 641, 643. FunctorPatternPtr: 941, 966, 967, 1061, 1097, fPos: 533, 551, 552, 601, 602, 608, 629, 630, 636, 1127, 1143, 1185, 1265. 640, 726, 729, 1496, 1499. FunctorSegment: 884, 887, 1081, 1143, 1191. fPriority: 562, 563. Functor Segment Obj: 886, 887. fpSignal: 95.FunctorSegmentPtr: 886, 1081, 1143, 1191, 1343. Fraenkel term: 784, 1053. FunctorSort: 941, 942. FraenkelTermObj: 784, 785.fVal: 533, 549, 550, 553. FraenkelTermPtr: 784, 1053, 1121, 1168, 1437. fVocFile: 557, 582.Free: 186, 188, 219, 222, 226. fW: 1500.FreeAll: 195, 200, 212, 220, 235, 240, 511. gAddSymbolsSet: 1495. FREEBSD: 81. *qAntecedent*: 1271, 1272, 1281. FreeExtItems: 235, 237, 249. gAttrColl: <u>1278</u>, 1279, 1281, 1398.

qBlockPtr: 687, 690, 694, 696, 699, 1203, 1223, GetEnvironment Variable: 45.1230, 1244, 1276, 1295, 1362, 1379, 1380, GetEnvironName: 52, 87. 1381, 1569, 1570, 1577, 1580, 1581, 1582, GetEnvStr: 44, 45, 484. 1583, 1584, 1585, 1587, 1588, 1589, 1590, 1591, GetExtension: 38. 1594, 1596, 1597, 1598, 1615, 1622, 1623, 1624, $GetFileExtName: \underline{49}, 50.$ 1631, 1632, 1633, 1634, 1635, 1637, 1638, 1640, GetFileName: 38, 48. 1643, 1647, 1648, 1649, 1650, 1651, 1652, 1653, GetFileTime: 34, 37, 471. 1654, 1657, 1659, 1661. $GetFuncPattern\colon \ \ \underline{1609},\ 1623,\ 1631,\ 1648.$ GCD: 17, 129, <u>152</u>, 153, 158, 159. GetIdentifier: 1199, 1201, 1285, 1289, 1310, 1325, qcd: 160.1335, 1342, 1347, 1349, 1353, 1357, 1361, 1362, gClusterSort: 1228, 1229, 1272, 1281, 1282. 1373, 1400, 1434, 1486. $gClusterTerm: \underline{1271}, 1272, 1282.$ GetIntAttr: 528, 532, 676, 677, 1012, 1108, 1110, gComment: 71, 73. 1111, 1114, 1117, 1118, 1119, 1120, 1121, 1123, gConsequent: 1271, 1272, 1281. 1125, 1126, 1127, 1128, 1131, 1133, 1134, 1136, gConstructorNr: <u>1268</u>, 1270, 1311, 1313, 1314, 1139, 1143, 1144, 1145, 1146. 1318, 1319, 1321, 1322, 1323, 1327, 1329, 1330. GetLocalTime: 120, 122, 123.aCorrCondSort: 1244, 1250. qetmem: 645.gCorrectnessConditions: 1240, 1244, 1249, 1250, GetMem: 189, 201, 215, 243, 252, 256, 277, 335, 1264, 1272, 1275, 1363, 1364, 1385. 338, 347, 417, 435, 496, 526. gDefiniens: 1228, 1229, 1249, 1264, 1265, 1266, GetMEOptions: 62, 68. 1267, 1385. GetMizFileName: 50, 51. $gDefining Way \colon \quad \underline{1227}, \ 1229, \ 1249, \ 1253, \ 1265,$ GetModePattern: 1604, 1624, 1631. 1299, 1300, 1303, 1304, 1385. GetNames: 464, 478.*qDefinitional*: 1374, 1375, 1376. GetObject: 195, 202, 207, 309, 323, 326, 329, 331. gDefKind: 1231, 1233, 1240. GetOptAttr: 528, 530. *gDefLabel*: 1367, 1368, 1385. GetOptions: 64, 69, 87. gDefLabId: 1297, 1299, 1368.GetPhrase: 604, 628, 629, 632, 633. gDefLabPos: <u>1297</u>, 1299, 1368. GetPredPattern: <u>1612</u>, 1623, 1631, 1632. gDefPos: 1240, <u>1241</u>, 1363. GetPrivateVoc: 557, 581.Generalization: <u>1568</u>, 1587, 1637. GetPublicVoc: 557, 582. geq: 141, 169. GetReferences: 1571, 1574, 1576. $qet_attribute$: 507. gets: 149, 150. $get_index_compare_to_default$: 1606. GetSchemeReference: 1574, 1576. $GetSortedNames: \underline{464}, \underline{477}.$ qet_taq_kind : 500. GetAccOptions: 61, 67. GetStdHandle: 98. GetAdjectiveCluster: 1523, 1530, 1548, 1644, GetString: 309, 322, 331. 1645, 1646. GetStructPatterns: 1615, 1623. GetArguments: 1505, 1509, 1510, 1519, 1526, GetTime: 120.1527, 1534, 1539, 1543, 1550. GetToken: 494, 495, 498, 503, 506, 507, 508, GetArticleName: 51, 52, 87. 509, 604, 629, 645, 726. GetAttr: 528, 531, 532, 676, 677, 1012, 1130, GetTransfOptions: 63, 70. 1142, 1143, 1144, 1145, 1146. GetVisible: 1602, 1603, 1607, 1608. GetAttrPattern: <u>1608</u>, 1623, 1631, 1632. gExpandable: 1232, 1233, 1242, 1264, 1307. GetAttrPos: 1097, 1100, 1108, 1110, 1111, 1112, gExpPtr: 685, 690, 705, 1351, 1519, 1530, 1531, 1113, 1114, 1115, 1116, 1117, 1118, 1119, 1120, 1549, 1561, 1643, 1644, 1645, 1646. 1121, 1123, 1125, 1126, 1127, 1128, 1130, 1131, gFieldsNbr: <u>1328</u>, 1329, 1330, 1333. 1133, 1134, 1136, 1138, 1139, 1140, 1141, 1143. gFormatsBase: 648, 1208. GetAttrValue: 494, 501, 507, 509, 1097, 1099. gFormatsColl: <u>648</u>, 680, 1208, 1311, 1314, 1319, 1323, 1327, 1329, 1330, 1333, 1395, 1409, 1418, GetClosedSubterm: 1508, 1519, 1520, 1531, 1549. GetConsoleMode: 98. 1443, 1445, 1447, 1449, 1462, 1463. GetDirectoryName: 38. gIdentifyEqLociList: 1274, 1275, 1283, 1285.GetEnv: 44. qInference: 1260, 1261, 1381.

```
qItemPtr: 686, 690, 698, 704, 1222, 1558, 1559,
                                                        qPartDef: 1302, 1303, 1304.
    1560, 1561, 1562, 1563, 1564, 1565, 1566,
                                                        gPartialDefs: 1228, 1229, 1303, 1304, 1385.
    1567, 1568, 1570, 1571, 1572, 1573, 1574,
                                                        gPattern: 1248, 1250, 1264, 1265, 1266, 1267,
    1575, 1576, 1578, 1579, 1581, 1582, 1583,
                                                            1275, 1277, 1283, 1311, 1314, 1319, 1323.
    1584, 1588, 1599, 1600, 1601, 1602, 1603, 1604,
                                                        gPatternPos: <u>1248</u>, 1264, 1265, 1266, 1267, 1270,
    1606, 1607, 1608, 1609, 1610, 1611, 1612, 1614,
                                                            1277, 1283, 1285, 1311, 1313, 1314, 1318, 1319,
    1615, 1616, 1617, 1619, 1620, 1621, 1622, 1623,
                                                            1321, 1322, 1323, 1327.
    1624, 1626, 1627, 1628, 1629, 1631, 1632, 1643,
                                                        gPremises: 1238, 1239, 1257, 1259, 1263, 1293,
    1644, 1645, 1646, 1647, 1648, 1649, 1651, 1652,
                                                            1294, 1296, 1359, 1360.
    1653, 1654, 1655, 1656.
                                                        gPriority: 648, 676, 679, 680, 1422, 1428.
gIterativeLastFormula: 1260, 1261, 1381.
                                                        qPrivateId: 1348, 1349, 1352, 1353, 1355, 1356,
qIterativeSteps: 1260, 1261, 1381, 1384.
                                                            1357, 1358, 1361, 1362.
gIterPos: 1382, 1383, 1384.
                                                        qPrivateIdPos: 1348, 1349, 1352, 1353, 1355,
qLastAdjective: 1394, 1395.
                                                            1356, 1357, 1358, 1361, 1362.
qLastFormula: 1211, 1212, 1213, 1229, 1250, 1259,
                                                        gProofCnt: 1208, <u>1210</u>, 1211, 1213, 1219, 1221,
    1261, 1300, 1303, 1304, 1344, 1345, 1356,
                                                            1458.
    1359, 1360, 1381, 1385, 1437, 1456, 1457,
                                                        gPropertySort: 1232, 1233, 1242, 1250, 1312.
    1458, 1462, 1463, 1464, 1465, 1466, 1467,
                                                        gQualifiedSegment: 1287, 1288, 1289, 1291.
    1469, 1470, 1471, 1473, 1474, 1475, 1477, 1478,
                                                        gQualifiedSegmentList: 1256, 1257, 1263, 1286,
    1479, 1480, 1487, 1488.
                                                            1291.
gLastTerm: 1227, 1229, 1275, 1282, 1285, 1300,
                                                        Grabowski, Adam: 12, 555, 647, 997.
    1303, 1352, 1355, 1357, 1361, 1362, 1384,
                                                        Graham, Paul: 792.
    1385, 1399, 1400, 1405, 1406, 1427, 1431,
                                                        Grammar, English: 1540.
    1437, 1439, 1441, 1443, 1445, 1447, 1449, 1451,
                                                        Grammar, for Mizar: 855.
    1452, 1453, 1454, 1455.
                                                        gReconsiderList: 1234, 1235, 1255, 1357, 1358.
gLastType: 1227, 1229, 1255, 1264, 1272, 1290,
                                                        gRedefinitions: 1207, 1217, 1243, 1244, 1249,
    1291, 1305, 1306, 1312, 1326, 1334, 1343, 1347,
                                                            1253, 1264, 1265, 1266, 1267, 1307, 1364, 1385.
    1350, 1391, 1402, 1407, 1409, 1410, 1411, 1435,
                                                        gResIdents: 1346, 1347.
    1436, 1451, 1466, 1484, 1485.
                                                        gResPos: 1346, 1347.
gLastWSBlock: 1202, 1203, 1207, 1211, 1213,
                                                        Grow: 309, 324, 328.
    1214, 1215, 1220, 1226, 1263, 1347, 1352,
                                                        GrowLimit \colon \  \, \underline{196},\, 204,\, 205,\, 238,\, 242,\, 254,\, 255,\, 276,\, \\
    1361, 1362.
                                                            345, 352, 353, 408, 415, 419, 433, 451, 461.
gLast WSItem: 1202, 1203, 1207, 1211, 1213, 1214,
                                                        gScanner: <u>723</u>, 726, 729, <u>730</u>, 731, 732, 733.
    1215, 1220, 1226, 1249, 1259, 1263, 1347,
                                                        gSchemeConclusion: 1248, 1250, 1344.
    1352, 1361, 1362.
                                                        gSchemeIdNr: 1248, 1250, 1335.
gLeftLoci: <u>1317</u>, 1318, 1319, 1322, 1323.
                                                        gSchemeIdPos: 1248, 1250, 1335.
gLeftLociNbr: 1316, 1318, 1319, 1322, 1323.
                                                        qSchemeParams: 1247, 1250, 1335, 1343.
qLeftLocus: 1284, 1285.
                                                        gSchemePos: 1245, 1246.
gLeft TermInReduction : \quad \underline{1273}, \ 1275, \ 1285.
                                                        gSchemePremises: <u>1248</u>, 1250, 1344, 1345.
gLocalScheme: \underline{1245}, \underline{1246}.
                                                        gSchVarIds: <u>1340</u>, 1341, 1342, 1343.
qLocus: 1309, 1310, 1311.
                                                        gSegmentPos: 1287, 1288, 1291.
qMainSet: 1496, 1497.
                                                        qSqmPos: 1331, 1332, 1334.
gMeansPos: 1252, 1253, 1299, 1385.
                                                        gSpecification: 1228, 1229, 1264, 1265, 1305, 1306.
gNewPattern: 1248, 1250, 1275, 1277, 1283.
                                                        gStartTime: 86, 117, 127.
gNewPatternPos: 1248, 1250, 1277, 1283, 1285.
                                                        gStructFields: 1269, 1270, 1329, 1334.
Gödel, Kurt: 922.
                                                        gStructFieldsSegment: 1331, 1332, 1333, 1334.
gOtherwise: 1298, 1299, 1300, 1385.
                                                        qStructPrefixes: 1241, 1242, 1270, 1326.
gParamNbr: 1308, 1310, 1311, 1313, 1314, 1315,
                                                        gSubexpPtr: 18, 684, 690, 712, 714, 715, 1427,
    1318, 1319, 1321, 1322, 1323, 1325, 1327, 1329.
                                                            1491, 1503, 1504, 1505, 1506, 1507, 1508,
gParams: <u>1269</u>, 1270, 1310, 1311, 1313, 1314,
                                                            1509, 1510, 1512, 1513, 1514, 1515, 1516,
    1315, 1318, 1319, 1321, 1322, 1323, 1324,
                                                            1517, 1518, 1519, 1520, 1521, 1522, 1523, 1524,
    1325, 1327.
                                                            1525, 1528, 1529, 1530, 1531, 1532, 1533, 1534,
```

```
1535, 1536, 1537, 1538, 1539, 1540, 1541, 1542,
                                                             IdentifierName: 1007, 1012, 1013.
     1543, 1544, 1545, 1547, 1548, 1549, 1550, 1551,
                                                             IdentifyProcessing: 61, 65, 66.
     1552, 1553, 1554, 1555, 1556, 1561, 1641, 1642,
                                                             IdentifyRegistrationObj: 997, 998.
     1643, 1644, 1645, 1646.
                                                             IdentifyRegistrationPtr: 997, 1094, 1143, 1191,
gSubExpPtr: 1541, 1547, 1548.
                                                                  1275.
gSubItemKind: 1320, 1321, 1322, 1323, 1337, 1343.
                                                             IdentLength: 615, 616, 622, 626.
gSubItemPos: <u>1338</u>, 1339, 1341, 1343.
                                                             IdentRepr: 1013, 1022, 1026, 1042, 1043, 1059,
qSuchPos: 1262, 1263, 1293.
                                                                  1067, 1070, 1073, 1078, 1157, 1164, 1166, 1167,
gSuchThatOcc: 1292, 1293.
                                                                  1170, 1175, 1178, 1183, 1190.
gt: 142, 152, 154, 155, 180.
                                                             .idx File: 733, 1012.
GT: 489, 500, 506, 507, 508, 509.
                                                             if: 10.
qThatPos: 1258, 1259, 1263, 1294, 1296.
                                                            if_def: 10.
gTHEFileNr: 1370, 1371, 1372, 1376, 1377.
                                                            if_not_def: 10.
                                                             IFDEF: 9, 10, 77, 80, 89, 92, 95, 96, 100, 101,
gTypeList: 1336, 1337, 1343, 1349, 1350, 1355,
     1356.
                                                                  119, 120, 128, 137, 139, 140, 141, 142, 143, 144,
                                                                  146, 147, 148, 149, 150, 151, 152, 153, 154, 155,
qVisibleNbr: 1602, 1603, 1606, 1608, 1613, 1618.
gWSTextProper: 1202, 1203, 1216.
                                                                  156, 157, 173, 174, 175, 176, 177, 187, 463.
                                                             IFNDEF: 10, 89, 95.
gWsTextProper: 1211, 1214, 1215, 1220, 1226,
     1263, 1347, 1352, 1361, 1362.
                                                             IgnoreProof: <u>1577</u>, 1578, 1600, 1657.
                                                             ikExplQualifiedSegm: 741, 746, 1025, 1113, 1121,
gXMLHeader: 514, 534.
                                                                  1143, 1159, 1168.
H: <u>116</u>, <u>124</u>, <u>126</u>.
halt: 46, 50, 71, 75, 108, 481.
                                                             ikImplQualifiedSegm: 741, 744, 1024, 1025, 1113,
Halt: 47, 90, 106.
                                                                  1121, 1143, 1159, 1168.
                                                             Im: 130, 131, 169, 170, 171, 173, 174, 175, 176,
Halt_: 90, 92.
                                                                  177, 179, 181.
HasInDom: 359, 370, 373, 382, 391, 392, 393,
                                                             Image: 187.
     429, 440, 441, 447, 458.
High: 17, 395, 1101.
                                                             implementation: 5.
                                                             Implicitly Qualified Segment Obj: 743, 744.
Hilbert's programme: 920.
                                                             Implicitly Qualified Segment Ptr: 743, 1014, 1024,
HowToDefine: 946, 966, 967, 1143, 1227.
                                                                  1025, 1097, 1112, 1148, 1158, 1159, 1168,
I: <u>26, 213, 245, 249, 252, 257, 259, 263, 264, 266,</u>
                                                                  1291, 1436, 1485.
     <u>278, 284, 285, 289, 290, 291, 306, 311, 316,</u>
                                                             in_AggrPattern: <u>1227</u>, 1229, 1329, 1447.
     317, 326, 334, 352, 356, 357, 363, 368, 369,
                                                             In\_Format: \underline{647}, 676, \underline{677}.
     <u>372</u>, <u>373</u>, <u>374</u>, <u>377</u>, <u>378</u>, <u>383</u>, <u>384</u>, <u>385</u>, <u>387</u>,
     403, 409, 424, 425, 433, 439, 440, 442, 443,
                                                             Inc: 579.
                                                             inc: 52, 106, 126, 150, 204, 223, 238, 242, 246,
     444, 451, 457, 458, 460.
                                                                  254, 255, 257, 266, 271, 272, 276, 278, 279, 289,
   <u>32, 50, 52, 53, 67, 68, 69, 136, 139, 148, 158</u>
                                                                  328, 334, 345, 352, 353, 363, 370, 375, 381, 383,
     <u>205</u>, <u>206</u>, <u>209</u>, <u>214</u>, <u>218</u>, <u>221</u>, <u>228</u>, <u>230</u>, <u>253</u>,
     346, 375, 381, 388, 390, 391, 393, 426, 434,
                                                                 388, 394, 396, 403, 408, 415, 419, 426, 433, 442,
                                                                  445, 446, 451, 498, 499, 500, 501, 538, 543, 577,
     <u>452</u>, <u>517</u>, <u>519</u>, <u>541</u>, <u>542</u>, <u>578</u>, <u>580</u>, <u>585</u>, <u>599</u>,
                                                                 616, 619, 627, 629, 635, 636, 1006, 1009, 1012,
     <u>605</u>, <u>633</u>, <u>1008</u>, <u>1016</u>, <u>1017</u>, <u>1018</u>, <u>1020</u>, <u>1025</u>,
                                                                  1146, 1194, 1211, 1219, 1325, 1333, 1405, 1427,
     1027, 1045, 1058, 1060, 1068, 1071, 1076, 1077,
                                                                  1429, 1498, 1503, 1528, 1577, 1602.
     <u>1080, 1095, 1153, 1154, 1156, 1159, 1160, 1161,</u>
                                                             include: 1143, 1249, 1264, 1272, 1275, 1364.
     <u>1165</u>, <u>1168</u>, <u>1169</u>, <u>1174</u>, <u>1176</u>, <u>1181</u>, <u>1184</u>, <u>1186</u>,
     1187, 1188, 1191, 1197.
                                                             IncorrectFormula: 848, 849.
ID: 489, 500, 503, 506, 507, 508, 509.
                                                             IncorrectFormulaPtr: 848, 1115, 1456, 1457, 1458.
Identification: \underline{1648}, \underline{1650}.
                                                             IncorrectTermObj: 794, 795.
                                                             IncorrectTermPtr: 794, 1121, 1399, 1447, 1453,
Identifier: 603, 605, 624, 724, 1201, 1362, 1365,
                                                                  1454. 1455.
     1368, 1369, 1502, 1507, 1509, 1515, 1519, 1520,
                                                             Incorrect Type Obj: 806, 807.
     1531, 1536, 1549, 1559, 1566, 1572, 1573, 1574,
     1580, 1581, 1582, 1583, 1584, 1588, 1602, 1607,
                                                             IncorrectTypePtr: 806, 1110, 1409, 1410.
     1608, 1609, 1612, 1613, 1626, 1628, 1630, 1643,
                                                             InCorrSentence: 1561, 1562.
     1647, 1648, 1651, 1654, 1655.
                                                             InCorrStatement: 1562, 1633.
```

Index: 195, 202, 219, 221, 222, 223, 224, 286, 293, 359, 365. IndexListPtr: 250, 256, 257, 267, 277. IndexOf: 18, 195, 209, 225, 250, 264, 286, 289, 309, 325, 326, 350, 356, 397, 401, 402, 414, 418, 422, 429, 439, 447, 457. IndexOfObject: 309, 329. IndexOfStr: 281, 284, 285, 1103, 1104, 1105, 1106. Index1: 309, 319. Index2: 309, 319. InferenceKind: 874, 875, 876, 877, 878. InferenceName: 1073, 1074, 1140. infError: 874, 1074, 1140, 1180, 1380. Infinite: 18. Infinitive: 562, 563, 564, 565, 567, 568, 570, 587. Infinitive: 1540. InfixFunctor: 941, 942, 1061, 1063, 1127, 1129, 1185. InfixFunctorPatternObj: 941, 942. InfixFunctorPatternPtr: 941, 1063, 1127, 1185, 1323. InfixOperatorSymbol: 724, 1322, 1323, 1502, 1515, 1519, 1520, 1521, 1531, 1532, 1545, 1549, 1580, 1609, 1610, 1630, 1631. InfixTermObj: 762, 763.InfixTermPtr: 762, 1047, 1121, 1168, 1427. info: 83, 89, 128, 182, 485, 487, 514, 647, 681, 850, 1199, 1492. Info: 309, 321, 341, 342, 402, 412, 520, 521, 524, 525. Info Char: 485. InfoCurPos: 485.InfoExitProc: 486. InfoFile: 485, 486, 529, 534. infofile: 137, 139, 140, 141, 142, 143, 144, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157. InfoInt: 485. InfoNewLine:<u>485</u>, 493. InfoPos: $\underline{485}$. InfoString: $\underline{485}$. InfoWord: 485. infProof: 874, 1074, 1140, 1180, 1378. infSchemeJustification: 874, 882, 1073, 1074, 1140, 1180, 1380. infSkippedProof: 874, 1074, 1140, 1180, 1219, infStraightforwardJustification: 874, 880, 1074, 1140, 1174, 1180, 1191, 1380. inherited: 200, 233, 234, 237, 239, 240, 241, 265, 268, 269, 287, 311, 351, 358, 380, 386, 398, 399, 404, 406, 407, 491, 495, 497, 503, 504,

536, 645, 646, 744, 746, 750, 752, 767, 769,

882, 887, 910, 912, 914, 918, 936, 938, 940, 942, 951, 955, 959, 961, 973, 975, 977, 981, 983, 985, 990, 992, 994, 1002, 1015, 1098, 1149, 1206, 1213, 1226, 1249, 1390, 1490. Init: 18, 186, 187, 191, 192, 195, 197, 198, 199, 219, 220, 229, 230, 231, 235, 236, 250, 251, 253, 267, 268, 281, 283, 284, 286, 287, 299, 300, 303, 305, 307, 308, 309, 310, 341, 343, 350, 351, 355, 359, 360, 361, 366, 367, 380, 398, 402, 404, 405, 406, 429, 431, 438, 447, 449, 456, 466, 467, 472, 474, 475, 476, 477, 478, 482, 490, 491, 495, 503, 513, <u>562</u>, 563, <u>571</u>, 572, <u>573</u>, 574, 576, 578, 583, <u>591</u>, 592, 594, 595, 597, <u>601</u>, 602, <u>604</u>, 608, 613, 614, 617, 624, 626, 627, 645, 651, 652, 653, 654, 655, 656, 657, 658, 665, 666, 667, 668, 669, 671, 672, 673, 674, 675, 676, 677, 693, 694, 698, 699, 703, 704, 705, <u>710</u>, 711, 712, <u>713</u>, 714, 727, <u>738</u>, 739, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, <u>751</u>, 752, <u>756</u>, 757, <u>758</u>, 759, <u>760</u>, 761, <u>762</u>, 763, <u>764</u>, 765, <u>766</u>, 767, <u>768</u>, 769, <u>770</u>, 771, <u>772</u>, 773, <u>774</u>, 775, <u>776</u>, 777, <u>778</u>, 779, <u>780</u>, 781, <u>782</u>. 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, <u>794</u>, 795, <u>798</u>, 799, <u>800</u>, 801, <u>802</u>, 803, <u>804</u>, 805, 806, 807, 810, 811, 812, 813, 814, 815, 816, 817, 818, 819, 820, 821, 822, 823, 824, 825, 826, 827, 828, 829, 830, 831, 832, 833, 834, 835, 836, 837, 838, 839, 840, 841, 842, 843, 844, 845, 846, 847, 848, 849, 856, 857, 858, 859, 860, 861, 862, 863, 864, 865, 868, 869, 871, 872, 875, 876, 877, 878, 879, 880, 881, 882, 884, 885, 886, 887, 888, 889, 890, 891, 893, 894, 895, 896, 897, 898, 900 901, 903, 904, 906, 907, 909, 910, 911, 912, 913, 914, <u>915</u>, <u>916</u>, <u>917</u>, <u>918</u>, <u>924</u>, <u>925</u>, <u>926</u>, <u>927</u>, <u>928</u>, 929, 930, 931, 933, 934, 935, 936, 937, 938, 939, 940, <u>941</u>, 942, <u>946</u>, 947, <u>948</u>, 949, <u>950</u>, 951, <u>952</u>, $953, \underline{954}, 955, \underline{956}, 957, \underline{958}, 959, \underline{960}, 961, \underline{962}$ 963, <u>964</u>, 965, <u>966</u>, 967, <u>968</u>, 969, <u>970</u>, 971, <u>972</u>, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000, 1001, 1002, 1003, 1004, 1101, 1107, 1108, 1109, 1110. 1111, 1112, 1113, 1114, 1115, 1116, 1117, 1118, 1119, 1120, 1121, 1122, 1123, 1124, 1125, 1126, 1127, 1128, 1130, 1131, 1132, 1133, 1134, 1135, 1136, 1137, 1138, 1139, 1140, 1141, 1142, 1143, 1144, 1145, 1146, 1198, 1203, <u>1205</u>, 1206, 1215, 1219, 1220, 1222, 1223, 1225, 1226, 1235, 1242, 1250, 1255, 1257, 1259, 1261, 1263, 1264, 1265,

1266, 1267, 1270, 1272, 1275, 1279, 1283, 1285,

1286, 1288, 1289, 1291, 1294, 1296, 1303, 1304,

773, 777, 779, 801, 803, 827, 829, 831, 833, 835, 837, 841, 843, 857, 859, 861, 878, 880,

1310, 1311, 1312, 1314, 1319, 1323, 1324, 1325, InsertElem: 359, 361, 368, 371, 397, 399, 1196. 1329, 1332, 1333, 1334, 1335, 1337, 1341, 1342, InsertExt: 235, 242, 267, 274, 276. 1343, 1344, 1345, 1347, 1349, 1351, 1352, 1355, ${\it InsertHiddenFalse:} \quad {\it 67}.$ 1356, 1357, 1358, 1359, 1360, 1361, 1362, 1365, InsertHiddenFiles: 61, 65, 66, 67. 1368, 1373, 1376, 1378, 1379, 1381, 1384, 1385, InsertIncorrBasic: 716, 718, 1456, 1544. 1388, 1390, 1391, 1393, 1395, 1399, 1400, 1402, InsertIncorrFormula: 716, 718, 1457, 1538, 1541, 1403, 1409, 1410, 1411, 1427, 1432, 1433, 1434, 1547, 1548, 1549, 1561. 1436, 1437, 1439, 1441, 1443, 1445, 1447, 1449, InsertIncorrTerm: <u>716</u>, 718, 1455, 1459, 1515, 1451, 1452, 1453, 1454, 1455, 1456, 1457, 1458, 1520, 1531. 1462, 1463, 1464, 1465, 1466, 1467, 1469, 1470, InsertIncorrType: 716, 718, 1410, 1525, 1643. 1471, 1478, 1479, 1480, 1481, 1482, 1483, 1485, InsertItem: 309, 312, 327, 328. 1486, 1487, 1488, 1489, 1490, 1491. InsertList: 195, 199, 206. $init_val_and_end$: 479. InsertObject: 309, 330, 331. InitAccOptions: 61, 65, 67. InsertPragma: 1193, 1196.InitCtrl: 94, 95, 97, 98. InsertTimes: 469, 471, 473. InitDisplayLine: 57, 71, 85. Instance: 1387, 1417, 1418, 1422, 1427, 1428, 1429. InitExitProc: 87, 89, 93. Integer: 583, 584, 585, 647, 659, 665, 675, 935, InitFile: 520, 522, 524, 526, 534, 1149. 937, 939, 1131, 1136, 1143, 1165, 1348, 1388. initialize_definition_item: 1240, 1242. integer: 24, 26, 30, 31, 32, 33, 45, 50, 52, 53, 57, InitLibrEnv: 463, 482. 58, 59, 67, 68, 69, 71, 73, 76, 85, 89, 95, 103, InitNatFunc: 379, 380, 381, 1198. 104, 105, 106, 107, 134, 135, 136, 139, 143, 144. InitNatSeq: 397, 398. 145, 146, 147, 148, 149, 150, 172, 180, 181, 184, InitParsing: 502, 503, 529. 195, 196, 197, 201, 202, 205, 206, 207, 208, 209, InitPass: 84, 85. 213, 214, 215, 216, 218, 219, 220, 221, 222, 223, InitProcessing: 84, 87. 224, 228, 230, 231, 235, 236, 243, 244, 245, 249, InitScannerNames: 1008, 1096, 1192. 250, 251, 252, 253, 254, 255, 256, 257, 259, 260, InitScanning: 494, 495, 503, 632, 645, 723, 263, 264, 265, 266, 267, 268, 270, 271, 272, 273, 730, 732. 277, 278, 279, 281, 282, 283, 284, 285, 286, 287, InitSorted: 250, 251, 267, 268, 283, 286, 287, 288, 289, 290, 291, 293, 294, 295, 296, 297, 299, 675, 676. 300, 302, 303, 304, 305, 306, 309, 310, 311, 312, InitSourceFile: 723, 730.315, 316, 317, 318, 319, 320, 321, 322, 323, 324, InitTokens: <u>593</u>, 594, 596. 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, InitWithElement: 359, 361. 335, 340, 341, 342, 343, 346, 347, 349, 350, 351, InitWSLookupTables: 1101, 1147. 352, 353, 354, 356, 357, 358, 359, 360, 361, 363, InitWsMizarArticle: 1199, 1203. 365, 368, 369, 370, 371, 372, 373, 374, 375, 377, InOutFileBuffSize: 488, 496, 497, 526, 527, 378, 379, 380, 381, 382, 383, 384, 385, 387, 388, 536, 539. 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, Insert: 195, 204, 206, 216, 218, 219, 227, 230, 231, 400, 401, 402, 403, 404, 408, 409, 411, 412, 413, 235, 238, 250, 255, 267, 274, 286, 290, 309, 327, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 330, 341, 345, 349, 350, 352, 359, 363, 371, 381, 424, 425, 426, 428, 429, 430, 431, 433, 434, 435, 382, 383, 387, 388, 399, 402, 408, 409, 418, 419, 437, 439, 440, 441, 442, 443, 444, 445, 446, 447, <u>429</u>, 433, 441, 442, <u>447</u>, 451, 459, 472, 474, 475, 448, 449, 451, 452, 453, 455, 457, 458, 459, 460, 476, 477, 478, 513, 516, 577, 578, 595, 597, 613, 461, 469, 470, 471, 473, 477, 478, 479, 485, 487, 614, 617, 624, 626, 627, 670, 671, 672, 673, 674, 493, 494, 501, 502, 505, 516, 517, 519, 520, 521, 676, 727, 1101, 1107, 1109, 1113, 1117, 1121, 524, 525, 528, 532, 533, 541, 542, 550, 558, 564, 1122, 1124, 1130, 1134, 1135, 1141, 1143, 1144, 578, 589, 591, 592, 593, 595, 596, 599, 600, 601, 1145, 1214, 1215, 1220, 1226, 1263, 1285, 1289, 602, 605, 612, 615, 618, 621, 625, 632, 633, 642, 1291, 1304, 1325, 1326, 1333, 1334, 1342, 1343, 648, 651, 652, 653, 654, 655, 656, 657, 658, 665, 1345, 1347, 1350, 1352, 1357, 1358, 1359, 1360, 666, 667, 668, 669, 670, 671, 672, 673, 674, 676, 1361, 1362, 1373, 1376, 1384, 1393, 1395, 1411, 677, 678, 679, 721, 727, 738, 739, 751, 752, 756, 1427, 1434, 1436, 1463, 1464, 1466, 1485, 1486. 757, 758, 759, 760, 761, 762, 763, 766, 767, 768, InsertD: 286, 291. 769, 772, 773, 774, 775, 776, 777, 778, 779, 780,

```
781, 800, 801, 802, 803, 810, 811, 812, 813, 818,
                                                              IntSet: 418, 419, 420, 421, 422, 423, 424, 425,
     819, 864, 865, 868, 869, 870, 871, 872, 881, 882,
                                                                   <u>426</u>, <u>427</u>, <u>428</u>.
     888, 889, 924, 925, 928, 929, 930, 931, 935, 936,
                                                              IntSetPtr: 418.
     937, 938, 939, 940, 941, 942, 1008, 1013, 1016,
                                                              IntToComplex: 134, 172.
     1017, 1018, 1020, 1025, 1027, 1045, 1058, 1060,
                                                              IntToHex: 516, 519.
     1068, 1071, 1076, 1077, 1080, 1095, 1100, 1103,
                                                              IntToStr: 30, 33, 91, 172, 550, 626.
     1104, 1105, 1106, 1108, 1110, 1111, 1114, 1115,
                                                              IntTriplet: 340, 429, 433, 435, 438, 441, 442.
     1118, 1119, 1120, 1121, 1123, 1125, 1126, 1127,
                                                              IntTripletList: 429.
     1128, 1133, 1134, 1139, 1141, 1146, 1148, 1152,
                                                              IntTripletListPtr: 429, 435.
     1153, 1154, 1156, 1159, 1160, 1161, 1168, 1169,
                                                              Int2PairOfInt: 447, 451, 456, 459.
     1174, 1176, 1181, 1184, 1186, 1187, 1188, 1191,
                                                              Int2PairOfIntFunc: 447, 448, 449, 450, 451, 452,
     1193, 1194, 1196, 1197, 1199, 1201, 1210, 1225,
                                                                   <u>453</u>, <u>454</u>, <u>455</u>, <u>456</u>, <u>457</u>, <u>458</u>, <u>459</u>, <u>460</u>.
     1248, 1249, 1268, 1291, 1297, 1308, 1311, 1314,
                                                              Int2PairOfIntFuncError: 447, 448, 451, 452, 460.
     1316, 1319, 1323, 1327, 1328, 1329, 1330, 1333,
                                                              Int2PairOfIntFuncPtr: 447.
     1370, 1387, 1388, 1389, 1393, 1395, 1404, 1409,
                                                              In WSMizFile Obj: 1097, 1098, 1099, 1100, 1107,
     1411, 1417, 1426, 1429, 1436, 1443, 1445, 1447,
                                                                   1108, 1109, 1110, 1111, 1112, 1113, 1114, 1115.
     1449, 1462, 1463, 1466, 1485, 1487, 1488, 1492,
                                                                   \underline{1118},\,\underline{1119},\,\underline{1120},\,\underline{1121},\,\underline{1122},\,\underline{1123},\,\underline{1124},\,\underline{1125},
     1496, 1498, 1500, 1501, 1503, 1505, 1506, 1518,
                                                                   1126, 1127, 1128, 1129, 1130, 1131, 1132, 1133,
     1519, 1520, 1525, 1531, 1533, 1534, 1539, 1543,
                                                                   <u>1134</u>, <u>1135</u>, <u>1136</u>, <u>1137</u>, <u>1138</u>, <u>1139</u>, <u>1140</u>, <u>1141</u>,
     1544, 1549, 1577, 1602, 1604, 1612, 1615.
                                                                   1142, 1143, 1144, 1145, 1146.
IntegerList: 402.
                                                              In WSMizFilePtr: 1097, 1147.
IntegerListPtr: 402, 403, 417.
                                                              IODEBUG: 92.
interface: 5.
                                                              IOResult: 86, 89, 90, 108.
InternalForgetfulFunctorTermObj: 780, 781.
                                                              ioresult: 46, 109, 576.
Internal Forget ful Functor Term Ptr: 780, 1052,
                                                              IOUtils: 25.
     1121, 1168.
                                                              is\_attr\_pattern: 1630.
Internal Selector Term Obj: 774, 775.
                                                              is\_attr\_token: 1641.
InternalSelectorTermPtr: 774, 1014, 1044, 1045,
                                                              is\_forgetful\_functor\_pattern: 1630.
     1097, 1120, 1168, 1447.
                                                              is\_inference\_error: 1380.
Intersect With: 359, 370.
                                                              is\_infix\_pattern: 1630.
IntKey: 299, 300, 301.
                                                              is\_predicate\_pattern: \underline{1630}.
IntListError: 402, 411, 412, 413, 415, 416, 423.
                                                              is\_selector\_pattern: 1630.
IntPair: 299, 304, 340, 341, 345, 347, 349, 350,
                                                              is\_space: 501.
     352, 353, 355, 359, 363, 364, 366, 371, 380,
                                                              IsEqualTo: 359, 374, 418, 425.
     382, 383, 388, 399, 447, 460.
                                                              IsEqWithInt: 133, 170, 176.
IntPairItem: 299, 303, 305.
                                                              IsIdentifierFirstLetter: 604, 616, 631.
IntPairItemPtr: 299, 304, 305, 306.
                                                              IsIdentifierLetter: 604, 616, 628, 631, 632, 644.
IntPairKeyCollection: 299, 304, 305, 306.
                                                              IsInSet: 418, 424, 428.
IntPairKeyCollectionPtr: 299.
                                                              IsintegerNumber: 133, 169.
IntPairList: 341.
                                                              IsMember: 350, 357, 358.
IntPairListPtr: 341, 347, 388.
                                                              IsMMLIdentifier: 50, 53.
IntPairSeq: 341, 342, 343, 344, 345, 346, 347,
                                                              IsMultipleOf: 379, 392.
     <u>348</u>, <u>349</u>, 350.
                                                              IsNaturalNumber: 133, 169.
IntPairSeqPtr: 341.
                                                              IsPrime: 129, 158, 169.
IntPtr: 299, 302.
                                                              IsPrimeNumber: 133, 169.
IntQuickSort: \underline{403}.
                                                              IsRational GEQ: 18.
IntRel: 350, 351, 352, 353, 354, 355, 356, 357,
                                                              IsRationalGT: 18, 133, \underline{171}.
     358, 359.
                                                              IsRationalLE: 133, 171.
IntRelPtr: 350.
                                                              IsSubsetOf: 359, 375, 376, 418, 426, 427.
                                                              IsSupersetOf: 359, 376, 418, 427.
Int Sequence: 402, \underline{404}, \underline{405}, \underline{406}, \underline{407}, \underline{408}, \underline{409}, \underline{410},
     <u>411</u>, <u>412</u>, <u>413</u>, <u>414</u>, <u>415</u>, <u>416</u>, <u>417</u>, 418, 554, 723.
                                                              IsValidSymbol: <u>558</u>, 560, 561, 568, 577.
                                                              it, reserved word: 792.
Int Sequence Ptr: 402.
```

it_Allowed: 1227, 1229, 1301, 1303, 1304, 1453. itAssumption: 702, 1079, 1084, 1142, 1143, 1191, 1239, 1259, 1263, 1587, 1637. itAttrAntonym: 702, 1079, 1080, 1142, 1143, 1191, 1242, 1250, 1632. itAttrSynonym: 702, 1079, 1080, 1142, 1143, 1191, 1242, 1250, 1631. $itAxiom: \ \ \underline{702},\ 1079,\ 1080,\ 1142,\ 1143,\ 1191,$ 1237, 1653. itCanceled: <u>702</u>, 1242, 1570. itCaseBlock: 702, 1079, 1086, 1142, 1143, 1191, 1211, 1246, 1251, 1594, 1596, 1597. itCaseHead: 702, 1079, 1084, 1142, 1143, 1191, 1237, 1259, 1596. itChoice: 702, 1079, 1080, 1142, 1143, 1191, 1237, 1257, 1580. itCluster: 702, 1079, 1093, 1142, 1143, 1191, $1242,\ 1272,\ 1643.$ itConclusion: <u>702</u>, 1079, 1080, 1142, 1143, 1191, 1211, 1220, 1237, 1250, 1587, 1589. itConstantDefinition: 702, 1079, 1084, 1142, 1143, 1191, 1226, 1246, 1352, 1583. itCorrCond: 702, 1079, 1086, 1142, 1143, 1191, 1211, 1244, 1250, 1622. itCorrectness: 702, 1079, 1087, 1142, 1143, 1191, 1211, 1244, 1250, 1622. itDefAttr: 702, 938, 1061, 1066, 1079, 1089, 1129, 1142, 1143, 1185, 1191, 1242, 1253, 1266, 1623. itDefFunc: 702, 942, 1061, 1079, 1090, 1129, 1142, 1143, 1185, 1191, 1233, 1242, 1253, 1265, 1385, 1623. itDefinition: 702, 1079, 1080, 1142, 1143, 1191, 1211, 1246, 1251, 1634, 1638, 1650. itDefMode: 702, 936, 1061, 1065, 1079, 1088, 1129, 1142, 1143, 1185, 1191, 1233, 1242, 1253, 1264, 1624. itDefPred: 702, 940, 1061, 1062, 1079, 1089, 1101, 1129, 1142, 1143, 1185, 1191, 1233, 1242, 1253, 1267, 1623. ItDefPred: 1105.itDefStruct: 702, 931, 1079, 1091, 1101, 1142, 1143, 1191, 1242, 1270, 1615. Item: 195, 211, 219, 223, 224, 225, 226, 286, 291, 292, 295, 298, 299, 301. ItemKind: 693, 698, 702, 703, 704, 856, 858, 860, 861, 933, 934, 968, 969, 1101, 1105, 1146, 1205, 1222, 1225, 1226, 1231. ItemLookUpTable: 1101.

ItemLookup Table: 1101, 1102, 1105.

ItemName: 1087, 1095, 1101, 1143.

ItemPtr: 690, 693, 698, 704.

ItemObj: 693, 703, 704, 705, 708, 1225.

Items: 195, 197, 198, 199, 201, 202, 204, 205, 206, 207, 209, 213, 214, 215, 216, 217, 218, 220, 221, 223, 224, 228, 230, 236, 238, 242, 243, 245, 249, 251, 252, 253, 254, 255, 256, 259, 261, 262, 266, 270, 271, 272, 273, 276, 277, 278, 279, 285, 289, 293, 305, 306, 341, 343, 345, 346, 347, 352, 353, 354, 355, 360, 363, 365, 366, 367, 368, 369, 370, 374, 375, 377, 380, 381, 383, 384, 385, 387, 388, 390, 391, 392, 393, 394, 395, 396, 400, 401, 471, 473, 513, 580, 587, 598, 599, 600, 606, 607, 618, 624, 630, 679, 1016, 1017, 1018, 1020, 1025, 1037, 1053, 1054, 1058, 1060, 1068, 1071, 1076, 1077, 1081, 1082, 1083, 1085, 1091, 1092, 1094, 1156, 1159, 1160, 1161, 1165, 1168, 1169, 1174, 1176, 1181, 1184, 1186, 1187, 1188, 1191, 1197, 1263, 1291, 1411, 1436, 1466, 1485, 1487, 1488. Iterative Equality Obj: 916, 918.IterativeEqualityPtr: 916, 1076, 1141, 1174, 1261. IterativeStepObj: 915, 917. IterativeStepPtr: 915, 1076, 1141, 1174, 1384. itExemplification: 702, 1079, 1086, 1142, 1143, 1191, 1226, 1246, 1251, 1361, 1362, 1588. itExistentialAssumption: 702, 1079, 1085, 1142, 1143, 1191, 1230, 1257, 1569. itFuncNotation: 702, 1079, 1080, 1142, 1143, 1191, 1242, 1250, 1631. itGeneralization: 702, 1079, 1084, 1142, 1143, 1191, 1226, 1230, 1250, 1587. itIdentify: 702, 1079, 1094, 1142, 1143, 1191, 1242, 1275, 1648. itIncorrItem: 702, 1080, 1105, 1142, 1143, 1191, 1580, 1631, 1632, 1633. itLociDeclaration: 702, 1079, 1084, 1142, 1143, 1191, 1226, 1242, 1250, 1637, 1640, 1650. itModeNotation: 702, 1079, 1080, 1142, 1143, 1191, 1242, 1250, 1631. itPerCases: 702, 1079, 1086, 1142, 1143, 1191, 1237, 1254, 1591. itPragma: 702, 1079, 1080, 1142, 1143, 1191, 1215, 1251. itPredAntonym: 702, 1079, 1080, 1142, 1143, 1191, 1250, 1632. itPredSynonym: 702, 1079, 1080, 1142, 1143, 1191, 1242, 1250, 1631. itPrivFuncDefinition: 702, 1079, 1084, 1142, 1143, 1191, 1246, 1251, 1581. itPrivPredDefinition: 702, 1079, 1084, 1142, 1143, 1191, 1246, 1251, 1582. itProperty: 702, 1079, 1087, 1142, 1143, 1191, 1211, 1233, 1250, 1623. itPropertyRegistration: 702, 1079, 1094, 1142, 1143, 1191, 1211, 1242, 1275, 1649.

itReconsider: 702, 1079, 1080, 1142, 1143, 1191, 1379, 1380, 1381, 1392, 1395, 1408, 1446, 1448, 1235, 1255, 1584. 1459, 1460, 1461, 1473, 1496, 1499, 1500, 1503, 1504, 1506, 1507, 1508, 1509, 1511, 1512, 1514, itReduction: 702, 1079, 1094, 1142, 1143, 1191, 1242, 1275, 1647. 1515, 1516, 1517, 1518, 1519, 1520, 1521, 1524, 1525, 1526, 1527, 1528, 1531, 1532, 1535, 1538, itRegularStatement: 702, 1079, 1080, 1142, 1143, 1191, 1211, 1237, 1250, 1580. 1539, 1540, 1541, 1543, 1544, 1545, 1546, 1547, itReservation: 702, 1079, 1080, 1142, 1143, 1191, 1548, 1549, 1550, 1552, 1553, 1554, 1555, 1556, 1559, 1564, 1570, 1571, 1572, 1573, 1574, 1575, 1226, 1246, 1251, 1347, 1651. 1576, 1577, 1578, 1579, 1580, 1584, 1585, 1586, itSchemeBlock: 702, 1079, 1080, 1142, 1143, 1191, 1587, 1588, 1589, 1590, 1591, 1592, 1593, 1595, 1211, 1246, 1251, 1654. 1596, 1598, 1599, 1604, 1605, 1607, 1608, 1609, itSchemeHead: 702, 1079, 1080, 1142, 1143, 1191, 1611, 1612, 1613, 1616, 1617, 1621, 1622, 1623, 1246, 1250, 1654. 1624, 1625, 1626, 1627, 1628, 1629, 1630, 1633, itSection: 702, 1079, 1080, 1142, 1143, 1191, 1214. 1634, 1635, 1636, 1637, 1638, 1639, 1640, 1641, $itSupposeHead: \ \ \underline{702},\ 1079,\ 1084,\ 1142,\ 1143,\ 1191,$ 1642, 1643, 1644, 1647, 1648, 1650, 1654, 1655, 1237, 1259, 1594, 1596. 1656, 1657, 1658, 1659, 1660, 1661, 1662. ItTermObj: 792, 793. $kind_is_radix_type$: <u>1524</u>. ItTermPtr: 792, 1121, 1453. kn: 1426, 1427. itTheorem: 702, 1079, 1080, 1142, 1143, 1191, Knuth, Donald E.: 5. 1211, 1237, 1250, 1652. Korniłowicz, Artur: 12, 555, 647, 997, 1001, 1003. *j*: <u>150</u>, <u>1411</u>, <u>1466</u>. L: 260, 270, 293, 320, 354, 365, 421, 437, 455. Join: 379, 387. *l*: <u>1404</u>, <u>1417</u>, <u>1429</u>. JoinAtom: 379, 389.LabelObj: 864, 865.Justification: 1578, 1622, 1623, 1649, 1652. JustificationObj: 875, 876, 877. LabelPtr: 864, 865, 909, 911, 912, 946, 947, 950, 951, 954, 955, 1014, 1067, 1097, 1130, JustificationPtr: 875, 913, 914, 915, 916, 917, 918, 1131, 1132, 1141, 1148, 1170, 1219, 1225, 978, 979, 980, 981, 982, 983, 984, 985, 1001, 1365, 1367, 1368. 1014, 1072, 1074, 1086, 1097, 1140, 1148, 1180, lArq: 1126.1191, 1219, 1225, 1260, 1378. lArgs: 1127, 1128. k: 31, 216, 370, 392, 394, 396, 428, 445, 446, 595, lArgsNbr: 677. <u>1249</u>, <u>1291</u>, <u>1393</u>, <u>1436</u>, <u>1485</u>, <u>1487</u>, <u>1488</u>. Last: 195, 203. $keep_eating_alpha digits$: 500. *lAtt*: 530, 531. $keep_getting_until_end_of_tag$: 500. *Latt*: **531**. Key: 286, 293. *lAttrExp*: 18, <u>1641</u>. KeyOf: 286, 289, 290, 291, 292, 293, 299, 301. lAttrNr: 1108. Key1: 286, 288, 295, 297, 299, 302, 304, 364, lCaseKind: 1591, 1593, 1594, 1596. 469, 470. LCM: 129, <u>153</u>, <u>379</u>, 381. Key2: 286, 288, 295, 297, 299, 302, 304, 364, lCnt: 1008, 1009, 1012. 469, 470. lCod: 1194.KillBlock: 687, 689, 1592, 1598, 1634, 1638, lCode: 479, 558, 560, 564, 566, 1100. 1650, 1654, 1658. lCol: 501, 1100. KillExpression: 685, 689, 1558, 1561, 1643, 1644, lConds: 1143.1645, 1646, 1651. KillItem: 686, 689, 1499, 1657. lCons: 1143.lConsoleMode: 98.KillSubexpression: <u>684</u>, 689, 1519, 1530, 1531, lConstructorNr: 1323.1549, 1552, 1553, 1554, 1555, 1556, 1557, 1561, 1643, 1644, 1645, 1646. *lCount*: $\underline{265}$, 388. lCounter: 1577.*Kind*: 562, 563, 564, 567, 577, 587, 589, 592, 595, 599, 600, 602, 605, 607, 608, 613, 629, 652, 654, lCStm: 1141.656, 658, 659, 661, 676, 677, 678, 721, 726, 729, lCurrChar: 612, 613, 616, 617, 619, 622, 624, 1201, 1217, 1218, 1237, 1239, 1246, 1283, 1293, 626, 627. 1311, 1321, 1322, 1327, 1337, 1362, 1363, 1364, lDct: 1008, 1009, 1012. 1365, 1366, 1368, 1369, 1375, 1376, 1377, 1378, lDef: 1143.

lDefiniensExpected: 1623, 1624. librenv: 463, 514, 588, 850. lDefKind: 1623, 1628. lIdent: 623, 624. lDenom: 177.*lIdNr*: 1139, 1141. lDict: 576, 577. Limit: 195, 197, 198, 199, 201, 204, 205, 215, 217, 220, 223, 230, 236, 238, 242, 243, 251, 252, 253, lDictLine: <u>568</u>, 576, 577. lDictName: 583. 254, 255, 256, 276, 277, 341, 343, 345, 347, 352, 353, 355, 360, 363, 366, 367, 380, 381, 388. *lDummy*: 578, 579, 583, 596, 597, 1008, 1009. le: 142, 151, 165.lIndent: 1187, 1191. *lIndex*: 255, 256, 277, 279, 419, 420, 618, 619, 620. LeadingZero: 125, 126.Line: 85, 86, 103, 104, 107, 108, 109, 485, 495, Lee, Gilbert: 920. 499, 552, 608, 636, 640, 641, 686, 687, 704, LeftBracketName: 1007, 1009, 1010, 1011, 1048, 729, 1017, 1073, 1079, 1095, 1100, 1139, 1144, 1064, 1168, 1185. LeftCircumfixSymbol: 724, 1321, 1323, 1502, 1511, 1145, 1146, 1191. 1515, 1519, 1520, 1531, 1549, 1580, 1609, 1630. LinearReasoning: <u>1586</u>, 1592, 1598, 1657. *lEl*: 493. lInFile: 1008, 1012, <u>1147</u>. lInfPos: 1139.length: 26, 28, 31, 32, 45, 53, 67, 68, 69, 70, 74, Link: 187. 80, 125, 136, 138, 139, 143, 144, 146, 147, 149, lInt: 181, 390, 426, 532, 727.150, 257, 338, 339, 482, 498, 499, 501, 516, 517, 518, 519, 534, 541, 559, 560, 561, 564, lIntPair: 349, 371, 382, 383.565, 566, 568, 577, 582, 614, 616, 619, 620, lIntStr: 567. 626, 633, 635, 636, 637, 638, 639, 640, 641, $lIntTriplet: \underline{441}, \underline{442}.$ 643, 1013, 1194, 1404, 1429. lInt2PairOfInt: 459. Length: 1387, 1413, 1422, 1428, 1430. LINUX: 81. lEnvFile: 600, 676, 679.ListError: 195, 202, 205, 207, 208, 221, 223, 224, leq: 139, 140, 141, 142, 144, 148, 149, 150, 238, 239, 240, 241, 245, 246, 264, 266, 268, 158, 168. 270, 273, 274, 275, 278, 279, 284. lEval: 389. ListQuickSort: 257, 259.LexemRec: 589, 591, 593, 595, 604, 605, 651, list1: 1393.676, 677. list2: 1393.lExp: 1385.lItem: 222.*lExpKind*: 1130, <u>1643</u>, 1644, 1645, <u>1647</u>. lItemKind: 1146.lExpr: 1130.lItems: 215, 243, 256, 277, 388.lExprKind: 1068.lKind: 558, 559, 578, 579, 596, 597, 1008, 1009, lFailed: 621, 622. 1010, 1130. lFields: 1143.ll: 1426, 1428. $l Field SgmPos \colon \ 1143.$ lLab: 1130, 1132, 1141. lFile: 479, 480, 583, 584. lLabId: 1131.lFileExt: 48. lLabPos: 1131.lFileNr: 1134.*lLeftArgs*: 1426, 1427, <u>1462</u>. lFormat: 666, 667, 668, 669. lLeftArgsNbr: 677, 1463. lLex: 676, 677. lFormatNr: 670, 671, 672, 673, 674, 1311, 1314, 1319, 1323, 1327, 1329, 1330, 1333, 1395, 1409, lLimit: 205, 388. <u>1443</u>, <u>1445</u>, <u>1447</u>, <u>1449</u>, 1462, 1463. lLIne: 582.lFrm: 1115, 1116, 1165, 1463. *lLine*: 479, 558, 559, 560, 561, 582, 1100. $lGcd: \underline{160}.$ lLinked: 1138. $Lh: \underline{74}.$ lLinkPos: 1138.lHexNr: 517, 518.*lList*: 201, 335, 417, 1110, 1113, 1115, 1117, 1121, LibraryPath: $\underline{464}$, $\underline{474}$. 1130, <u>1135</u>, <u>1143</u>, 1393. lLocus: 1143.LibraryReferenceKind: 720, 722. Library Reference Name: 722.lMizarReleaseNbr: 479, 481. LibraryReferenceObj: 870, 871. lMizarVariantNbr: 479, 481. LibraryReferencePtr: 870. lMizarVersionNbr: 479, 481.

lModeMaxArgsSize: 727. lParenthCnt: 1520, 1521, 1522, 1525, 1528, 1529, <u>1531</u>, 1532, <u>1549</u>, 1551. lModesymbol: 1604.lModeSymbol: 1110, 1606.lPattern: 1143. $lName: 39, 40, 41, 42, \underline{45}, 472, \underline{581}.$ *lPos*: 479, 558, 560, 561, <u>564</u>, 565, 566, 612, 613, 614, 617, 622, 624, 626, 627, 1108, 1110, 1111, lNbr: 578, 579. <u>1112</u>, <u>1113</u>, <u>1114</u>, <u>1115</u>, 1116, 1117, <u>1118</u>, <u>1119</u>, lNeg: 1165. 1120, 1121, 1123, 1125, 1126, 1127, 1128, 1130, lNoneOcc: 1108.<u>1132</u>, <u>1133</u>, <u>1134</u>, 1136, <u>1138</u>, 1139, <u>1140</u>, <u>1141</u>. *lNr*: 596, 597, 1008, 1009, 1010, 1012, <u>1103</u>, <u>1104</u>, 1143, <u>1144</u>, <u>1145</u>, 1146, <u>1498</u>. 1105, 1106, 1111, 1114, 1115, 1117, 1118, 1119, lPredMaxArgsSize: 727. 1120, 1121, 1123, 1125, 1126, 1127, 1128, 1133, lPredSymbol: 1544, 1612, 1613.1134, 1136, 1139, 1143. lPrf: 727.lNumber: 625, 626. lPrintWhere: 1168.Load: <u>562</u>, 568, 578. *lPriority*: 558, 560. LoadDct: 593, 596, 645. $lProp: \underline{1137}.$ LoadFIL: 469, 472. lPropertySort: 1143.LoadFormats: 665, 676, 1208. lpszTempPath: 45. $LoadMmlVcb: \underline{557}, \underline{583}.$ lRedefinition: 1143.LoadPrf: 723, 727, 732. lRefPtr: 1376.Load Voc: <u>573</u>, 578, 582, 583. lRel: 80. lObj: 1099.lRepr: 564, 565, 566. lObject: 189. $lRes: \ \underline{161}, \ \underline{162}, \ \underline{163}, \ \underline{164}, \ \underline{165}, \ \underline{172}, \ \underline{173}, \ \underline{174},$ LocalReference: 867, 869, 1070, 1071, 1134, 1176. <u>175</u>, <u>176</u>, 177, <u>294</u>. LocalReferenceObj: 868, 869. $lResult \colon \quad \underline{312}, \, \underline{313}, \, \underline{315}, \, \underline{320}, \, \underline{325}, \, \underline{329}, \, \underline{401}, \, \underline{414}, \, \underline{422}.$ LocalReferencePtr: 868, 1014, 1070, 1097, 1133, lRightArgs: 1426, 1427, 1462, <u>1463</u>. 1148, 1175, 1373. lRightSymNr: 677. LocFilesCollection: 465, 474, 482. *lRNr*: 1121, 1127, 1128. LociEqualityObj: 995, 996. lRPos: 1121, 1127, 1128. LociEqualityPtr: 995, 1094, 1143, 1191, 1285. lSegment: 1436.LocusObj: 928, 929. lSels: 1143.LocusPtr: 928, 937, 938, 995, 996, 1014, 1059, lSqm: 1115.1097, 1123, 1126, 1143, 1148, 1183, 1284, lShape: 1130, 1146. 1285, 1309, 1310, 1325. lSpelling: 621, 622, 624, 626. LongInt: 466, 467. lStartTaqNbr: 1146. Longint: 34, 37, 135, 180, 294, 461. $lStr: 31, 32, \underline{33}, 45, 80, \underline{81}, \underline{125}, \underline{567}, \underline{1194}.$ longint: 83, 116, 117, 121, 122, 123, 124, 126, lString: 596, 597, 1008, 1009, 1010, 1012. 133, 170, 472, 520, 625. lStringObj: 284.LongLines: 64, 67, 68, 69, 636. lStructModeMaxArgsSize: 727. $LookUp_BracketFormat: \underline{665}, 668, 671, 1443.$ *lStructureSymbol*: <u>1615</u>, 1617, 1618. LookUp_FuncFormat: 665, 667, 672, 1418. lSymbNbr: 578, 579. LookUp_PredFormat: 665, 669, 674, 1462, 1463. lSymbol: 576, 577, 1525, 1526, 1527. LookUp_PrefixFormat: 665, 666, 673, 1395, 1409, LT: 489, 500, 503, 506, 507, 508, 509. 1445, 1447, 1449. lTemp: 403.Look UpFuncFormat: 1417.lTime: 472.LOOP programming language: 922. lTimeStr: 126. $lOption: \underline{70}.$ lToken: 595, 618, 619, 620. lOtherwise: 1130.lTrm: 1115, 1121, 1141, 1143, 1426, 1427. Low: 17, 1101. lType: 1143.LowerCase: 30, 32.lTyps: 1143.lPair: 399.lValue: 676.lpairItem: 305.lVar: 80, 1113, 1143. lPairItem: 305.lVars: 1143.

lVer: 80.MFormatPtr: 647, 651, 659, 660, 661, 665, lVer1: 479, 481. 670, 677, 679. lVer2: 479, 481. MFormatsList: 648, 665, 666, 667, 668, 669, 670, lWSMizOutput: 1096, 1192.<u>671</u>, <u>672</u>, <u>673</u>, <u>674</u>, <u>675</u>, <u>676</u>, <u>679</u>, 1417. MakeEnv: 68. MFormatsListPtr: 665. Mark: 235, 244. MIndexList: 250.MInfixFormatObj: 655, 656, 657, 667, 669. MaxArgListNbr: 1390, 1459. MInfixFormatPtr: 655, 664, 672, 674, 677, 678. MaxCollectionSize: <u>184</u>, 194, 215, 243, 250, 256, 277. Misses: <u>359</u>, 377, <u>418</u>, 428. MaxConstInt: 588, 626. $missing_functor_format: 1418, 1419, 1420, 1423,$ MaxIntegerListSize: 184, 402. 1424, 1425. MaxIntPairSize: 340, 341, 345, 347, 352. Missing Word: 1498, 1499, 1500, 1525, 1532, 1538, MaxIntTripletSize: 340, 429, 435. 1544, 1545, 1573, 1575, 1594, 1597. MaxLineLength: 588, 636, 637. $MItemList: \underline{194}.$ MaxListSize: <u>184</u>, 309, 335, 417. MizarExitProc: 85, 86, 87. MaxSize: 184, 340. *MizAssert*: 113, 492, 510, 531, 613, 666, 676. MaxSubTermNbr: 1199, 1208. mizassert: 1013, 1211, 1226, 1411, 1466. MaxVisArgNbr: 682, 1505, 1509, 1510, 1514, Mizassert: 105, 505, 1500. 1519, 1521, 1532, 1545, 1550. mizenv: 23, 55, 83, 89, 102, 128, 463, 485, 514, 554, 588, 719, 850, 1193, 1199. MBracketFormatObj: 657, 658, 668.MBracketFormatPtr: 657, 663, 671, 677, 678. $MizExitProc: \underline{92}, \underline{93}.$ MCollection: 219, 220, 221, 222, 223, 224, 225, MizFileName: 24, 50, 52, 87, 482, 486, 534, <u>226</u>, <u>227</u>, <u>228</u>, <u>229</u>, <u>230</u>, <u>231</u>, <u>232</u>, <u>233</u>, <u>234</u>, 1009, 1012. 286, 295, 573, 604. Mizfiles: 483.mconsole: 25, <u>54</u>, 83, 89, 102, 463, 485, 588, MizFiles: 464, 474, 480, 481, 482, 483, 484, 535, 600, 1015. 681, 850, 1199. *MCopy*: 186, 190, 195, 201, 206, 253. MizLoCase: 30, 32.MizPath: 464, 482, 484. mdebug: $\underline{9}$. MDEBUG: 9.MList: 195, 196, 197, 198, 199, 200, 201, 202, 203, MDuplicates: 309.204, 205, 206, 207, 208, 209, 210, 211, 212, 213, *MError*: <u>85</u>, 87. 214, 215, 216, 217, 218, 219, 233, 234, 235, Metamathematics: 920.250, 252, 253, 1287, 1336, 1340, 1388. $MList.AppendTo: \underline{216}.$ MExtList: 235, 236, 237, 238, 239, 240, 241, 242, MList.At: 202.243, 244, 245, 246, 247, 248, 249, 267. MExtList.AddExtItems: 247.MList.AtInsert: 205.MExtList.AddExtObject: 246.MList.CopyItems: 218.MExtList.DeleteAll: 239. MList.CopyList: 199. MExtList.DeleteExtItems: MList.DeleteAll: 210.248. MExtList.Done: 237. MList.Done: 200.MList.FreeAll: 212.MExtList.FreeAll: 240. MExtList.FreeExtItems: 249.Mlist.FreeItem: 211.MExtList.FreeItemsFrom: 245.MList.FreeItemsFrom:MExtList.Init: 236.MList.GetObject: 207.MExtList.Insert: 238. MList.ListError: 209. MExtList.InsertExt: 242. $MList.Init: \underline{197}.$ MExtList.Mark: 244. $MList.Insert: \underline{204}.$ MExtList.Pack: 241.MList.InsertList:MExtList.SetLimit: 243.MList.Last: 203.MExtListPtr: 235. MList.ListError: 208. Meyer, Albert R.: 922. MList.MCopy: 201.MFileExists: 34, 35, 36, 47, 474, 581. MList.MoveList: 198.MFormatObj: 651, 652, 653, 678. $MList.Pack: \underline{214}.$

MList.SetLimit: 215.MSortedCollection: 18, 286, 287, 288, 289, 290, 291, 292, 293, 295, 299, 305, 469. MList. TransferItems: 217.MListPtr: 195.MSortedCollection.Compare: 288.MSortedCollection.IndexOf:MML: 464, 480. MSortedCollection.Init: 287.MMLIdentifier: 724, 1572, 1573, 1574. MSortedCollection.InitSorted:287.MMLIdentifierName: 1007, 1009, 1010, 1071, MSortedCollection.Insert: 290.1073, 1176, 1179. $MSortedExtList\colon \ \ 18,\, \underline{267},\, \underline{268},\, \underline{269},\, \underline{270},\, \underline{271},\, \underline{272},$ *MObject*: <u>186</u>, <u>187</u>, <u>188</u>, <u>189</u>, <u>190</u>, 191, 195, 197, 273, 274, 275, 276, 277, 278, 279, 280. 198, 199, 201, 213, 220, 236, 245, 251, 253, 299, $MSortedExtList.Done: \underline{269}.$ 307, 309, 310, 311, 341, 343, 360, 402, 429, 431, MSortedExtList.Find:447, 449, 466, 494, 520, 524, 562, 571, 651, 710, MSortedExtList.Z: 267.738, 741, 747, 753, 797, 808, 856, 860, 862, 864, MSortedList: 250, 251, 252, 253, 254, 255, 256, 867, 875, 884, 888, 890, 893, 895, 897, 900, 903, 259, 260, 263, 264, 265, 266, 281, 665. 906, 909, 915, 924, 926, 928, 930, 946, 948, 952, MSortedList.CopyList: 253.956, 962, 964, 966, 970, 978, 984, 987, 1003. MSortedList.Find: 260. *mObject*: 864, 933, 968, 995, 997, 999. MSortedList.FreeItemsFrom:mobjects: <u>182</u>, 462, 463, 487, 514, 554, 588, 647, MSortedList.IndexOf: 264. 681, 719, 735, 850, 1193, 1199. MSortedList.Init: 251. Mod: 129, 151. MSortedList.InitSorted: 251. ModeDefinitionObj: 956, 957, 958, 960. MSortedList.Insert: 255.ModeDefinitionPtr: 956, 1088, 1191.MSortedList.MoveList: 252. ModeDefinitionSort: 956, 957. MSortedList.Pack: 265.ModeDefinitionSortName: 1088, 1143. MSortedList.Search: 263. ModeMaxArgs: 723, 727, 728, 1526, 1606. MSortedList.SetLimit:ModeName: 1007, 1009, 1010, 1011, 1021, 1065, MSortedList.Sort: 259.1162, 1185. MSortedList.Z: 254. ModePatternObj: 935, 936. MSortedStrList: 281, 283, 284, 285, 502, 530, ModePatternPtr: 930, 931, 935, 956, 957, 958, 593, 1101. 959, 960, 961, 1065, 1097, 1125, 1143, 1185, MSortedStrList.IndexOfStr: 284. 1264, 1314. MSortedStrList.Init: 283. ModeSymbol: 724, 1408, 1409, 1515, 1524, 1525, MSortedStrList.ObjectOf: 285. 1548, 1604, 1630, 1631, 1641. mstate: 83.monitor: 83, 89. MStringCollection: 295, 297, 298, 464, 475, 477. Move: 189, 215, 223, 243, 254, 256, 277, 279, 318, MStringItem: 309, 311, 318, 319, 328, 335. 328, 335, 347, 352, 353, 355, 363, 366, 380, 411, MStringItemList: 309. 415, 417, 419, 433, 435, 438, 451, 456. MStringList: 17, 309, 310, 311, 312, 315, 316, 317, MoveCollection: 219, 229. 318, 319, 320, 321, 322, 323, 324, 325, 326, MoveList: 195, 198, 219, 233, 250, 252, 1291, 1343, 327, 328, 329, 330, 331, 332, 333, 334, 335, 1355, 1356, 1398, 1437, 1439, 1465, 1467, 1485. 336, 337, 557, 583, 584, 585. MoveNatFunc: 379, 380.MStringList.AddObject: 315. MoveNatSet: 359, 367.MStringList.AddString:MoveObject: 309, 331. MStringList.AddStrings: 316. MoveSequence: 402, 406. MStringList.Clear: 317. MoveStringList: 309, 310.MStringList.Delete: 318.MPrefixFormatObj: 653, 654, 655, 666. MStringList.Done: 311. MPrefixFormatPtr: 653, 662, 673, 677, 678. MStringList.Exchange: 319. mscanner: 719, 850, 1199, 1492. MStringList.ExchangeItems: 319. *MScannObj*: 632, 633, 642, 643, 644, 645, 646. MStringList.Find: 320. MScannPtr: 632, 723, 732. MStringList.GetObject: 323. Msg1: 59, 74.MStringList.GetString: 322. Msq2: 59, 74. MStringList.Grow: 324.

MStringList.IndexOf: 325. nArticleID: 856, 857. MStringList.IndexOfObject: 329. nArticleNr: 870, 872, 1071, 1176. MStringList.Insert: 327. nAssumptionPos: 970, 971, 1092. MStringList.InsertItem: 328. nAssumptionSort: 970, 971, 1092, 1182. MStringList.ObjectOf: 326. NatFunc: 379, 380, 381, 382, 383, 384, 385, MStringList.PutObject: 333. <u>386, 387, 388, 389, 391, 392, 393, 394, 396,</u> MStringList.PutString:397, 461, 1193. 332.MStringList.QuickSort: 334. NatFuncPtr: 379, 388, 389, 390. NatSeq: 397, 398, 399, 400, 401. MStringList.SetCapacity: 335. MStringList.SetSorted: 336. NatSet: 359, 360, 361, 362, 363, 365, 366, 367, MStringList.Sort: 337. <u>368</u>, <u>369</u>, <u>370</u>, <u>371</u>, <u>372</u>, <u>373</u>, <u>374</u>, <u>375</u>, <u>376</u>, <u>377</u>, <u>378</u>, 379, 1193. MStringList.StringListError: 321. MStringList.Z: 329. NatSetError: 341, 342, 345, 346, 352, 353, MStrObj: 191, 192, 284, 490, 491, 591. 363, 384, 385. MStrPtr: 191, 1101. NatSetPtr: 359.nAttrCollection: 1388, 1391, 1395, 1398, 1411, mtime: 83, 115. MTokeniser: 604, 605, 608, 609, 613, 628, 629, 1466, 1467. 631, 632. nAttrSymbol: 937, 938, 1066, 1185. MTokenObj: 601, 602, 604. nAttrVals: 502, 503, 504, 511, 513, 530, 531, MTokenPtr: 601, 613, 614, 617, 624, 626, 627, 630. 1099, 1100. Naumowicz, Adam: 12, 555, 647, 997. Mul: 17, 129, 147, 153, 156, 158, 161, 162, 163, nBlock: 860, 861, 1080, 1086, 1087, 1094, 1095, 164, 165, 168. mul: 148, 150, 151. 1146, 1191, 1211. MultiPredicativeFormulaObj: 814, 815. nBlockEndPos: 856, 859, 1017, 1145, 1213. MultiPredicativeFormulaPtr: 814, 1037, 1117, nBlockKind: 693, 694, 856, 859, 1017, 1187, 1206, 1211, 1213. 1165, 1465. nBlockPos: 856, 859, 1016, 1017, 1144, 1145, 1216. mul1: 146, 147. Mul1: 146. nChoiceType: 790, 791, 1057, 1168. $MUnpackTime: 116, \underline{124}, 126.$ nClusterKind: 987, 988, 1093, 1191. nClusterPos: 987, 988, 1093. nAdjectiveCluster: 804, 805, 1021, 1162, 1411, nClusterTerm: 993, 994, 1093, 1191. 1466. *nAdjectivePos*: 747, 748, 1019. nClusterType: 987, 988, 990, 994, 1093, 1191. nConditionalDefiniensList: 954, 955, 1068, 1186. nAdjectives: 816, 817, 1038, 1165. nAdjectiveSort: 747, 748, 1019, 1155. nConditions: 906, 907, 974, 975, 977, 982, 983, 1082, 1085, 1087, 1092, 1143, 1182, 1191. nAdjectiveSymbol: 751, 752, 1019, 1155. nAllPos: 1388, 1438, 1439. nConnective: 1388, 1473, 1478. NamesFile: 475, 476, 477, 478. nConsequent: 987, 988, 1093, 1191. nAncestors: 930, 931, 1091, 1191. nConsistent: 379, 380, 382, 383, 384, 386, 387, 388, 389, 398. nAntecedent: 991, 992, 1093, 1191. nContent: 860, 861, 1079, 1080, 1081, 1082, 1083, nArg: 749, 750, 770, 771, 773, 779, 822, 823, 1084, 1085, 1086, 1087, 1088, 1089, 1090, 1091, 937, 938, 1019, 1028, 1050, 1051, 1066, 1155, 1092, 1093, 1094, 1095, 1142, 1143, 1146, 1191, 1165, 1168, 1185, 1463. 1215, 1220, 1250, 1254, 1255, 1257, 1259, 1261, nArgList: <u>1388</u>, 1390, 1413, 1422, 1427, 1428, 1429, 1430, 1459. 1263, 1264, 1265, 1266, 1267, 1270, 1272, 1275, 1312, 1347, 1352, 1355, 1356, 1361, 1362. nArgListNbr: 1388, 1390, 1413, 1418, 1419, 1420, 1421, 1422, 1423, 1427, 1428, 1429, 1430, 1459. nCorrCondPos: 978, 979.nCorrCondSort: 980, 981, 1079, 1191. nArgs: 751, 752, 764, 765, 767, 769, 777, 798, 799, nCurCol: 494, 495, 498, 499, 500, 501. 818, 819, 935, 936, 937, 938, 941, 942, 1019, nCurTokenKind: 494, 499, 500, 503, 505, 506, 1021, 1026, 1043, 1048, 1049, 1064, 1065, 1066, 1091, 1155, 1162, 1164, 1167, 1168, 1185, 1191. 507, 508, 509. nArticleExt: 856, 857, 1016. nDefAttrPattern: 962, 963, 1089, 1191. nArticleId: 1016.nDefAttrPos: 962, 963.

nDefFuncPattern: 966, 967, 1090, 1191. 1452, 1453, 1454, 1455, 1456, 1457, 1458, 1462, 1463, 1465, 1466, 1467, 1469, 1470, 1471, 1478, nDefFuncPos: 966, 967.1479, 1480, 1485, 1486, 1487, 1488, 1491. nDefiniens: 960, 961, 962, 963, 964, 965, 966, 967, 1088, 1089, 1090, 1191. new_Bl: 1417, 1424. nDefiniensProhibited: <u>1228</u>, 1229, 1253, 1307. NewAccom: 62, 68. nDefiningWay: 966, 967, 1079, 1191. NewBlock: 856, 858, 1211. nDefKind: 956, 957, 1088, 1191. NewIndex: 309, 331.NewItem: 856, 858, 1214, 1215, 1220, 1226, 1263, nDefLabel: 946, 947, 1068, 1186. 1347, 1352, 1361, 1362. nDefModePattern: 956, 957, 1088, 1191. nDefModePos: 956, 957.NewSignal: 95.NewStr: 328, 332, 338, 461, 467, 475, 476, nDefNr: 871, 872, 1071. 477, 478. nDEfNr: 1176.nDefPos: 946, 947, 1068. nExpansion: 958, 959, 1088, 1191. nExpKind: 710, 711. nDefPredPattern: 964, 965, 1089, 1191. nExpr: 948, 949, 1068, 1186. nDefPredPos: 964, 965.nExpression: 950, 951, 1068, 1186. nDefSort: 946, 947, 1068, 1186, 1249. nExprKind: 948, 949, 1068, 1186. nDefStructPattern: 930, 931, 1091, 1191. nDisplayInformationOnScreen: 1014, 1015, 1079, $next_term_has_higher_precedence$: 1422. NextElementState: 502, 508, 510, 676, 677, 1012, 1097, 1098, 1144, 1145, 1146, 1148, 1149, 1191. 1108, 1109, 1110, 1111, 1112, 1113, 1114, NegatedAdjectiveObj: 749, 750.1115, 1116, 1117, 1118, 1119, 1120, 1121, 1122, NegatedAdjectivePtr: 749, 1019, 1108, 1155, 1395. 1123, 1124, 1125, 1126, 1127, 1128, 1130, 1131, NegativeFormulaObj: 822, 823. NegativeFormulaPtr: 822, 1028, 1115, 1165, 1463, 1132, 1133, 1134, 1135, 1138, 1139, 1140, 1141, 1143, 1144, 1145, 1146. 1466, 1467, 1471. NextTag: 502, 506.nElName: 502, 503, 504, 511, 676, 1012, 1108, nFieldPos: 924, 925, 1091. 1110, 1113, 1115, 1121, 1127, 1129, 1130, 1131, nFields: 926, 927, 1091, 1191. 1134, 1135, 1140, 1141, 1143, 1144, 1145, 1146. nFieldSegmPos: 926, 927, 1091. nelName: 1138.nFieldSymbol: 924, 925, 1091, 1191. Nelson, Alexander M: great sense of humour: 618. *nFile*: 520, 522, 523, 524, 526, 534, 536, 537, 539, nEqLociList: 997, 998, 1094, 1143, 1191. 1149, 1150, 1151, 1152, 1153. nEqPos: 995, 996, 1094. nFileBuff: 524, 526, 527. New: 474, 859. nFirstSententialOperand: 1388, 1473, 1474, 1475,new: 472, 513, 522, 577, 578, 581, 582, 583, 585, 1478, 1479, 1480. 595, 597, 613, 614, 617, 624, 626, 627, 671, 672, nFormula: 784, 785, 1053, 1168. 673, 674, 676, 677, 698, 699, 705, 712, 730, 732, nFormulaPos: 808, 811, 813, 815, 817, 819, 821, 858, 878, 931, 1012, 1096, 1101, 1107, 1108, 823, 825, 839, 845, 847, 849, 1026, 1027, 1028, 1109, 1110, 1111, 1112, 1113, 1114, 1115, 1116, 1029, 1030, 1031, 1032, 1033, 1034, 1035, 1036, 1117, 1118, 1119, 1120, 1121, 1122, 1123, 1124, 1037, 1038, 1039, 1040, 1041. 1125, 1126, 1127, 1128, 1130, 1131, 1132, 1133, nFormulaSort: 808, 811, 813, 815, 817, 819, 821, 1134, 1135, 1136, 1137, 1138, 1139, 1140, 1141, 823, 827, 829, 831, 833, 835, 837, 841, 843, 1142, 1143, 1144, 1145, 1146, 1147, 1192, 1203, 845, 847, 1027, 1163, 1165, 1463. 1215, 1219, 1220, 1222, 1223, 1235, 1242, 1250, nformula Sort: 849.1255, 1257, 1259, 1261, 1263, 1264, 1265, 1266, *nFunc*: 1388, 1390, 1417, 1418, 1419, 1420, 1422, 1267, 1270, 1272, 1275, 1279, 1283, 1285, 1286, 1423, 1424, 1425, 1427, 1428, 1429. 1289, 1291, 1294, 1296, 1303, 1304, 1310, 1311, 1312, 1314, 1319, 1323, 1324, 1325, 1329, 1332, nFuncId: 893, 894, 1084, 1191. nFunctKind: 941, 942, 1061, 1185. 1333, 1334, 1335, 1342, 1343, 1344, 1345, 1347, nFunctorIdent: 768, 769, 1043, 1167. 1351, 1352, 1355, 1356, 1357, 1358, 1359, 1360, nFunctorSymbol: 762, 763, 1047, 1168. 1361, 1362, 1365, 1368, 1373, 1376, 1378, 1379, nGuard: 952, 953, 1068, 1186. 1381, 1384, 1385, 1393, 1395, 1398, 1399, 1400, 1402, 1403, 1409, 1410, 1411, 1427, 1434, 1436, nHasAssumptions: 1205, 1207, 1230, 1244, 1295. 1437, 1439, 1441, 1443, 1445, 1447, 1449, 1451, *nIdent*: 738, 739, 756, 757, 1022, 1042, 1157, 1166. nIdentifier: 743, 744, 1024, 1158, 1168. nLine: 494, 495, 497, 498, 499, 500, 501. nIdentifiers: 745, 746, 890, 891, 1025, 1077, 1159, nLinkable: 1225, 1229, 1237, 1239, 1276.1168, 1191, 1436. nLinkAllowed: 1205, 1207, 1214, 1215, 1276, 1380. nIdentifyPos: 997, 998. nLinked: 879, 880, 1072, 1074, 1174, 1191, 1205, nIndent: 533, 534, 542, 543, 546, 548, 1148, 1207, 1218, 1276, 1379, 1380, 1381. 1149, 1154, 1187, 1191. nLinkPos: 879, 880, 1072, 1205, 1207, 1218, 1276, 1379. nInDiffuse: 1205, 1208, 1211, 1362. *nInference*: <u>1225</u>, 1250, 1254, 1255, 1257, 1261, nLinkProhibited: 1205, 1207, 1214, 1215, 1218, 1275, 1369, 1373, 1376, 1377, 1378, 1379, 1276. nLocusNr: 758, 759, 1046, 1168. 1380, 1381, 1384. nInfPos: 875, 876, 1073, 1074. nMizarAppearance: 1014, 1015, 1019, 1021, 1022, nInfSort: 875, 876, 1074, 1174, 1180, 1191, 1380. 1026, 1035, 1036, 1042, 1043, 1044, 1046, nItAllowed: 1225, 1227, 1229, 1242, 1301, 1304. 1047, 1048, 1049, 1050, 1051, 1059, 1062, nItemEndPos: 860, 861, 1095, 1146, 1249, 1263, 1063, 1064, 1065, 1066, 1067, 1070, 1071, 1347, 1352, 1361, 1362. 1073, 1078, 1091, 1096. nModeKind: 1388, 1408, 1409.nItemKind: 703, 704, 860, 861, 1079, 1080, 1095, nModeNr: 1388, 1408, 1409. 1142, 1143, 1191, 1211, 1226, 1240, 1250, nModeSymbol: 800, 801, 935, 936, 1021, 1065, 1253, 1263, 1385. nItemPos: 860, 861, 1079, 1095, 1191, 1225, 1091, 1162, 1185, 1191. 1229, 1250, 1257, 1259, 1272, 1275, 1312, nMultipredicateList: 1388.nMultiPredicateList: 1463, 1464, 1465. 1352, 1361, 1362. *nName*: 466, 467, 468, 470, 471, 473. *nItems*: 856, 859, 1016, 1017, 1144, 1145, 1187, nNegated: 751.1188, 1214, 1215, 1220, 1226, 1263, 1347, 1352, 1361, 1362. nNewPattern: 968, 969, 997, 998, 1080, 1094, nIterPos: 915, 917, 1076. 1191. nIterSteps: 916, 918, 1076, 1141, 1174. nNewTerm: 1003, 1004, 1094, 1191. nJustification: 900, 901, 906, 907, 913, 914, 915, nNextWord: 1388, 1446, 1447, 1448. nNoneOcc: 1388, 1392, 1395.917, 978, 979, 984, 985, 1001, 1002, 1075, 1076, 1082, 1083, 1086, 1087, 1094, 1141, 1143, nNonPos: 1388, 1392, 1395.1172, 1174, 1191, 1275. nNotationPos: 968, 969. *nLab*: 864, 865, 909, 912, 1069, 1076, 1171, nNotationSort: 968, 969. 1174, 1191. nNotPos: 1388, 1466, 1467, 1472. *nLabel*: 1225, 1250, 1259, 1261, 1345, 1359, $no_longer_referencing_article$: 1573. NoDisplayLine: 57, 58, 71. 1360, 1365. nLabelIdNr: 864, 865, 1067, 1170, 1186, $\underline{1225}$, Noise: 59, 71, 74. 1365. NonBlockReasoning: <u>1591</u>, 1592, 1598, 1657. nLabelIdPos: 1225, 1365.nOperSymb: 941, 942, 1063, 1185. nLabelPos: 864, 865, 1067. nOriginPattern: 968, 969, 997, 998, 1080, 1094, nLabId: 868, 869, 1070, 1175. 1191. nLastSentence: 1205, 1211, 1213. nOriginTerm: 1003, 1004, 1094, 1191.NotationBlock: 1638, 1660. nLastWSBlock: 1205, 1207, 1213. nLastWSItem: 1205, 1207, 1213, 1214, 1215, 1225, NotationDeclarationObj: 968, 969. 1226, 1250, 1254, 1255, 1257, 1261, 1263, 1264, NotationDeclarationPtr: 968, 1080, 1143, 1191, 1265, 1266, 1267, 1270, 1272, 1275, 1312, 1347, 1250. 1352, 1355, 1356, 1361, 1362. NotationsProcessing: 61, 65, 66, 67. nLeftArg: 824, 825, 1029, 1030, 1031, 1032, nOtherwise: 954, 955, 1068, 1186. 1033, 1034, 1163. NotPrivate: 1549.nLeftArgs: 762, 763, 812, 813, 939, 940, 941, 942, nPartDefiniens: 952, 953, 1068, 1186. 1035, 1047, 1062, 1063, 1165, 1168, 1185. nPatternPos: 933, 934, 1062, 1063, 1064, 1065, nLeftBracketSymb: 941, 942, 1064, 1185. 1066. nLeftBracketSymbol: 766, 767, 1048, 1168. nPatternSort: 931, 933, 934, 1061, 1185. nLeftLocus: 995, 996, 1094, 1191. *nPos*: 494, 495, 498, 499.

nPostNegated: 1388, 1459, 1466, 1467, 1472. nPostqualification: 782, 783, 785, 1053, 1054, nPostQualList: 1388, 1432, 1436, 1437, 1439. nPragmaStr: 862, 863, 1079, 1191. *nPredId*: 895, 896, 1084, 1191. nPredIdNr: 818, 819, 1026, 1164. nPredNr: 810, 811, 813, 1035, 1036, 1165. nPredSymbol: 939, 940, 1062, 1185. nProp: 913, 914, 972, 973, 1075, 1092, 1141, 1172, 1182, 1191. nPropertyPos: 984, 985, 999, 1000. nPropertySort: 984, 985, 999, 1000, 1079, 1094, 1191. nPropPos: 1225, 1250, 1259, 1261, 1280, 1345, 1359, 1360. nQualification: 786, 787, 1055, 1168. nQualifiedSegments: 1388, 1481, 1482, 1485, 1487, 1488. nQualVars: 906, 907, 1082, 1191.nQuaPos: 1388, 1401, 1402, 1403. nQVars: 976, 977, 1085, 1143, 1191. Nr: 49, 589, 592, 595, 599, 600, 602, 605, 606, 607, 613, 624, 652, 654, 656, 658, 660, 676, 677, 678, 721, 726, 729, 1201, 1233, 1242, 1244, 1311, 1313, 1318, 1321, 1322, 1323, 1327, 1333, 1365, 1368, 1369, 1371, 1372, 1375, 1376, 1377, 1395, 1408, 1429, 1440, 1442, 1443, 1444, 1446, 1448, 1452, 1454, 1460, 1461, 1468, 1496, 1526, 1527, 1540, 1541, 1546, 1547, 1573, 1575, 1606, 1613, 1617, 1649. nRedefinition: 956, 957, 962, 963, 964, 965, 966, 967, 1088, 1089, 1090, 1191. nReducePos: 1003, 1004.nReferences: 877, 878, 1073, 1074, 1138, 1139, 1177, 1179, 1373, 1376. nRefPos: 867, 869, 872, 1070, 1071. nRefSort: 867, 869, 872, 1071, 1176. nRegularStatementKind: 1225, 1261, 1366, 1381. nRestriction: 1388, 1390, 1477, 1488. nRestrPos: 1388, 1476, 1488. nResType: 890, 891, 1077, 1191. nRightArg: 824, 825, 1029, 1030, 1031, 1032, 1033, 1034, 1163. nRightArgBase: 1388, 1397, 1459, 1460, 1461, 1462, 1463. nRightArgs: 762, 763, 810, 811, 813, 939, 940, 941, 942, 1035, 1036, 1047, 1062, 1063, 1165, 1168, 1185, 1463. nRightBracketSymb: 941, 942, 1064, 1185. nRightBracketSymbol: 766, 767, 1048, 1168. nRightLocus: 995, 996, 1094, 1191.

nRightSideOfPredPos: 1388, 1461. nRSymbolNr: 1388, 1443. nSample: 782, 783, 785, 1053, 1054, 1168, 1388, 1431, 1437, 1439. nSchemeConclusion: 888, 889, 1081, 1143, 1191. nSchemeIdNr: 881, 882, 888, 889, 1073, 1078, 1178, 1179, 1190, 1369, 1377, 1380. nSchemeInfPos: 881, 882, 1073, 1139, 1369, 1377. nSchemeParams: 888, 889, 1081, 1143, 1191. nSchemePos: 888, 889. nSchemePremises: 888, 889, 1081, 1143, 1191. nSchFileNr: 881, 882, 1073, 1179, 1377. nScope: 838, 839, 1040, 1041, 1165. nScraps: 814, 815, 1037, 1165. nSegment: 838, 839, 1040, 1041, 1165. nSegmentIdentColl: 1388, 1433, 1434, 1436, 1483, 1485, 1486. nSegmentPos: <u>1388</u>, 1435, 1436, 1483, 1485. nSegmentSort: 741, 742, 1025, 1159, 1168. nSegmPos: 741, 742, 884, 885, 1024, 1025, 1081, 1263. 1487. 1488. nSegmSort: 884, 885, 1081, 1191. nSelectorSymbol: 772, 773, 774, 775, 1044, 1050, 1168. nSentence: 864, 865, 895, 896, 1069, 1084, 1171, 1191. nSethoodType: 1001, 1002, 1094, 1191. nSgmFields: 930, 931, 1091, 1191. nSntPos: 864, 865. nSourceFile: 494, 496, 497, 499. nSourceFileBuff: 494, 496, 497.nSpecification: 886, 887, 926, 927, 960, 961, 966, 967, 1081, 1088, 1090, 1091, 1191. nSpelling: 494, 495, 497, 498, 499, 501, 503, 511, 513, 1388, 1440, 1441, 1468, 1469. nState: 502, 506, 508, 509, 510, 676, 1012, 1107, 1109, 1113, 1117, 1121, 1122, 1124, 1127, 1130, 1131, 1134, 1135, 1140, 1141, 1143, 1144, 1145, 1146. nStatementSort: 909, 910, 912, 918, 1076, 1079, 1174. nStrPos: 930, 931, 1091. nStructSymbol: 776, 777, 778, 779, 780, 781, 802, 803, 1021, 1049, 1051, 1052, 1162, 1168. nSubexpPos: 1388, 1397, 1408, 1409, 1440, 1441, 1446, 1447, 1448, 1449, 1450, 1451, 1459, 1460, 1462, 1463, 1465, 1466, 1468, 1469, 1473, 1478, 1481, 1482, 1487, 1488. nSubExpPos: 1467, 1478.nSubject: 787, 788, 789, 816, 817, 820, 821, 1038,

1039, 1055, 1056, 1165, 1168.

nSymbolNr: 1388, 1442, 1443, 1444, 1445, 1446, 1447, 1448, 1449, 1459, 1460, 1461, 1462, 1463. *nTerm*: 915, 917, 1076, 1174. nTermBase: 1388, 1390, 1395, 1396, 1399, 1409, 1413, 1441, 1443, 1445, 1456, 1459, 1462, 1468, 1469. nTermExpr: 893, 894, 897, 898, 900, 901, 903, 904, 1083, 1084, 1086, 1191. nTermPos: 753, 757, 759, 761, 763, 765, 771, 775, 781, 783, 785, 787, 789, 791, 793, 795, 1042, 1043, 1044, 1045, 1046, 1047, 1048, 1049, 1050, 1051, 1052, 1053, 1054, 1055, 1056, 1057. nTermSort: 753, 757, 759, 761, 763, 765, 771, 775, 781, 783, 785, 787, 789, 791, 793, 795, 1045, 1168. nTheoNr: 871, 872, 1071, 1176. nType: 745, 746, 804, 805, 820, 821, 1021, 1025, $1039,\ 1159,\ 1162,\ 1165,\ 1168.$ nTypeChangeKind: 900, 901, 1083, 1191. nTypeChangeList: 900, 901, 1083, 1191. nTypeExpList: 884, 885, 893, 894, 895, 896, 1081, 1084, 1191. nTypeExpr: 900, 901, 1083, 1191. nTypePos: 797, 799, 805, 807, 1021, 1411, 1466. nTypeSort: 797, 799, 805, 807, 1021, 1162. Num: 130, 131, 160, 161, 162, 163, 164, 165, 167, 168, 169, 170, 171, 172, 177, 181. numbers: 128, 182. Numeral: 603, 605, 626, 724, 1376, 1377, 1502, 1508, 1515, 1519, 1520, 1531, 1549, 1570, 1573, 1575, 1580. NumeralTermObj: 760, 761.NumeralTermPtr: 760, 1045, 1121, 1168, 1452. nValue: 490, 491, 513, 530, 531, 760, 761, 1045, 1099, 1100, 1168. nVar: 900, 901, 1083, 1191. nVarId: 897, 898, 903, 904, 928, 929, 1059, 1084, 1086, 1183, 1191. nVarIdPos: 928, 929, 1059.nVarPos: 738, 739, 1022, 1291, 1436, 1485. nVars: 884, 885, 1081, 1191. object: 6. *Object*: 217. ObjectOf: 17, 281, 285, 299, 305, 309, 326, 530, 531, 1099, 1100. ObjectPtr: 186.Occurs: 1500, 1507, 1515, 1533, 1535, 1536, 1563, 1565, 1566, 1567, 1568, 1579, 1583, 1584, 1588, 1603, 1606, 1607, 1608, 1617, 1619, 1620, 1626, 1627, 1628, 1629, 1635, 1648, 1651, 1655. OldSigInt: 95.OneArgumentTermObj: 770, 771, 772, 778.

One Argument Term Ptr: 770.OpenedErrors: 107, 108, 109, 110, 111. OpenErrors: 87, 105, 108. OpenFile: 528, 529, 533, 534, 535, 585, 600, 676, 679, 1012, <u>1014</u>, 1015, 1096, <u>1097</u>, 1098, 1147, 1148, 1149, 1192. OpenFileWithXSL: 533, 535, 1014, 1015. OpenInfoFile: 87, 485, 486. OpenParenth: 1503, 1520, 1531, 1549.OpenStartTag: 502, 506, 509, 511. Ord: 32, 516, 519, 1101, 1105. ord: 622, 626, 676, 1005, 1006, 1101, 1375, 1422, 1428, 1573, 1575. Other: 370. $other_regular_statements$: <u>1580</u>. othercases: 8. others: 8. Out_Adjective: 1014, 1019, 1020. Out_AdjectiveList: 1014, 1020, 1021, 1038, 1093. Out_Block: 1014, 1017, 1074, 1076, 1080, 1086, 1095.Out_CompactStatement: 1014, 1075, 1076, 1080. Out_Definiens: 1014, 1068, 1088, 1089, 1090. $Out_Format: \ \ \underline{651}, \ 678, \ 679.$ Out_Formula: 1014, 1027, 1028, 1029, 1030, 1031, 1032, 1033, 1034, 1037, 1040, 1041, 1053, 1068, 1069, 1081, 1084. Out_ImplicitlyQualifiedVariable: 1014, 1024, 1025. $Out_InternalSelectorTerm: 1014, 1044, 1045.$ Out_Item: 1014, 1016, 1017, 1095. Out_ItemContents: 1014, 1080, 1095. $Out_ItemContentsAttr: 1014, 1079, 1095.$ Out_Justification: 1014, 1074, 1075, 1076, 1082, 1083, 1086, 1087, 1094. $Out_Label\colon \ \ \underline{1014},\ 1067,\ 1068,\ 1069,\ 1076.$ Out_Link: 1014, 1072, 1074. Out_LocalReference: 1014, 1070, 1071. Out_Loci: 1014, 1060, 1062, 1063, 1064, 1065, 1066, 1091. Out_Locus: 1014, 1059, 1060, 1066, 1094. $Out_NumReq2$: 175. Out_NumReq3: 173, 174, 176, 177. Out_Pattern: 1014, 1061, 1080, 1088, 1089, 1090, 1094. Out_PosAsAttrs: 533, 551, 552, 1016, 1017, 1019, 1021, 1022, 1024, 1025, 1026, 1028, 1029, 1030, 1031, 1032, 1033, 1034, 1035, 1036, 1037, 1038, 1039, 1040, 1041, 1042, 1043, 1044, 1045, 1046, 1047, 1048, 1049, 1050, 1051, 1052, 1053, 1054, 1055, 1056, 1057, 1059, 1062, 1063, 1064, 1065, 1066, 1067, 1068, 1070, 1071, 1072, 1073, 1074, 1076, 1081, 1091, 1092, 1093, 1094, 1095.

586 INDEX §1665 Mizar Parser

Out_PrivateFunctorTerm: 1014, 1043, 1045. Out_PrivatePredicativeFormula: 1014, 1026, 1027. Out_Proposition: 1014, 1069, 1075, 1081, 1082, 1085, 1092. Out_References: 1014, 1071, 1073, 1074. Out_RegularStatement: 1014, 1076, 1080. Out_ReservationSegment: 1014, 1077, 1080. Out_Reserved Variable: 1014, 1023, 1077. Out_SchemeJustification: 1014, 1073, 1074. Out_SchemeNameInSchemeHead: 1014, 1078, 1081. Out_SimpleTerm: 1014, 1042, 1045. Out_Term: 1014, 1018, 1038, 1039, 1045, 1050, 1051, 1053, 1054, 1055, 1056, 1068, 1076, 1083, 1084, 1086, 1093, 1094. Out_TermList: 1014, 1018, 1019, 1021, 1026, 1035, 1036, 1043, 1047, 1048, 1049. Out_TextProper: 1014, 1016, 1096. Out_Type: 1014, 1021, 1025, 1039, 1055, 1057, 1058, 1077, 1081, 1083, 1088, 1090, 1091, 1093, 1094. Out_TypeList: 1014, 1058, 1081, 1084. Out_Variable: 1014, 1022, 1023, 1024, 1025, 1081, 1083, 1084, 1086. Out_VariableSegment: 1014, 1025, 1040, 1041, 1053, 1054, 1082, 1084, 1085. Out_XAttr: 533, 549, 550, 553, 585, 586, 587, 600, 678, 679, 1016, 1017, 1019, 1021, 1022, 1026, 1035, 1036, 1042, 1043, 1044, 1046, 1047, 1048, 1049, 1050, 1051, 1059, 1062, 1063, 1064, 1065, 1066, 1067, 1068, 1070, 1071, 1073, 1078, 1079, 1087, 1091, 1095. Out_XAttrEnd: 533, 544, 545, 585, 600, 1016, 1017, 1019, 1020, 1021, 1024, 1025, 1026, 1028, 1029, 1030, 1031, 1032, 1033, 1034, 1035, 1036, 1037, 1038, 1039, 1040, 1041, 1043, 1047, 1048, 1049, 1050, 1051, 1053, 1054, 1055, 1056, 1057, 1062, 1063, 1064, 1065, 1066, 1068, 1069, 1073, 1074, 1076, 1081, 1091, 1092, 1093, 1094, 1095. Out_XElEnd: 533, 548, 585, 587, 600, 679, 1016, 1017, 1019, 1020, 1021, 1024, 1025, 1026, 1028, 1029, 1030, 1031, 1032, 1033, 1034, $1035,\ 1036,\ 1037,\ 1038,\ 1039,\ 1040,\ 1041,$ 1043, 1047, 1048, 1049, 1050, 1051, 1053, 1054, 1055, 1056, 1057, 1058, 1060, 1062, 1063, 1064, 1065, 1066, 1068, 1069, 1073, 1074, 1076, 1077, 1081, 1082, 1083, 1085, 1087, 1088, 1090, 1091, 1092, 1093, 1094, 1095. Out_XElEnd0: 533, 546, 547, 551, 586, 587, 600, 678, 679, 1019, 1020, 1021, 1022, 1026, 1042, 1043, 1044, 1045, 1046, 1048, 1049, 1052, 1059, 1067, 1070, 1071, 1072, 1074, 1081, 1087, 1091. paColonExp3: 1494, 1654.

Out_XElStart: 533, 543, 545, 547, 551, 585, 586, 587, 600, 678, 679, 1016, 1017, 1019, 1020, 1021, 1022, 1024, 1025, 1026, 1028, 1029, 1030, 1031, 1032, 1033, 1034, 1035, 1036, 1037, 1038, 1039, 1040, 1041, 1042, 1043, 1044, 1045, 1046, 1047, 1048, 1049, 1050, 1051, 1052, 1053, 1054, 1055, 1056, 1057, 1059, 1062, 1063, 1064, 1065, 1066, 1067, 1068, 1069, 1070, 1071, 1072, 1073, 1074, 1076, 1081, 1087, 1091, 1092, 1093, 1094, 1095. Out_XElStart0: 533, 545, 585, 587, 679, 1025, 1035, 1036, 1047, 1058, 1060, 1068, 1077, 1081, 1082, 1083, 1085, 1087, 1088, 1090, 1091, 1092. Out_XElWithPos: 533, 551, 1021, 1027, 1045, 1074.Out_XEl1: 533, 547, 1035, 1036, 1045, 1047, 1060, 1088, 1089, 1090. Out_XIntAttr: 533, 550, 552, 586, 587, 600, 678, 679, 1017, 1019, 1021, 1022, 1026, 1035, 1036, 1042, 1043, 1044, 1045, 1046, 1047, 1048, 1049, 1050, 1051, 1052, 1059, 1062, 1063, 1064, 1065, 1066, 1067, 1070, 1071, 1073, 1078, 1091, 1095. $Out_XQuotedAttr:$ 533, 553, 600. OutChar: 533, 538, 540, 541, 542, 543, 544, 548. 549. OutIndent: 533, 542, 543, 548. OutNewLine: 533, 540, 544, 546, 548. OutString: 533, 534, 535, 541, 543, 546, 548, 549. 1015. OutWSMizFileObj: 1014, 1015, 1016, 1017, 1018, 1019, 1020, 1021, 1022, 1023, 1024, 1025, 1026. 1027, 1042, 1043, 1044, 1045, 1058, 1059, 1060, 1061, 1067, 1068, 1069, 1070, 1071, 1072, 1073, 1074, 1075, 1076, 1077, 1078, 1079, 1080, 1095. OutWSMizFilePtr: 1014, 1096.OverflowErroor: 90, 91, 103, 112. OverflowError: 105, 112. P: 338.paAdjClusterExp: 1494, 1644, 1645. paAllExp: 1494, 1517. paArrowExp1: 1494, 1614.paArrowExp2: 1494, 1644. paAsExp: 1494, 1584. paAttrExp1: 1494, 1524. paAttrExp2: 1494, 1608. paAttrExp3: 1494, 1641. paCasesExp: 1494, 1593. Pack: 195, 214, 219, 228, 235, 241, 250, 265, 267, 275. paColonExp1: 1494, 1513.paColonExp2: 1494, 1626, 1628.

 $\S1665$ Mizar Parser INDEX 587

	Diela Demonal Em 10. 1404 1520
paColonExp4: 1494, 1573, 1575.	paRightParenthExp10: 1494, 1532.
paDefExp: 1494, 1573.	paRightParenthExp11: 1494, 1608.
paEndExp: 1494, 1499.	paRightParenthExp2: 1494, 1509.
paEqualityExp1: 1494, 1581, 1648.	paRightParenthExp3: 1494, 1519.
paEqualityExp2: 1494, 1583.	paRightParenthExp4: 1494, 1551.
paForExp: 1494, 1631, 1632, 1645, 1651.	paRightParenthExp5: 1494, 1607.
paForOrArrowExpected: 1494, 1643.	paRightParenthExp6: 1494, 1616.
paFunctExp1: 1494, 1545.	paRightParenthExp7: 1494, 1574.
paFunctExp2: 1494, 1610.	paRightParenthExp8: 1494, 1581.
paFunctExp3: 1494, 1532.	paRightParenthExp9: 1494, 1655.
paFunctExp4: 1494, 1643, 1647.	paRight Square Exp1: 1494.
paIdentExp1: 1494, 1507.	paRightSquareExp2: 1494, 1550.
paIdentExp10: 1494, 1626, 1628.	paRight Square Exp3: 1494.
paIdentExp11: 1494, 1651.	paRightSquareExp4: 1494, 1582.
paIdentExp12: 1494.	paRightSquareExp5: 1494, 1655.
paIdentExp13: 1494, 1655.	Parse: 1492, 1658.
paIdentExp2: 1494, 1536.	parse_comma_separated_types: <u>1579</u> .
paIdentExp3: 1494, 1602, 1648.	$parse_proof: \underline{1578}.$
paIdentExp4: 1494, 1566.	parser: 1199, <u>1492</u> .
paIdentExp5: 1494.	parseraddition: $\underline{1199}$.
paIdentExp6: 1494, 1581.	ParserOnly: 64, 69.
paIdentExp7: 1494, 1581.	PartDefObj: 952, 953.
paIdentExp7: 1494, 1582. paIdentExp8: 1494, 1583.	PartDefPtr: 952, 1068, 1130, 1186, 1304.
7	
paIdentExp9: 1494, 1584.	paSchExp: 1494, 1575.
paIfExp: 1494, 1627, 1629.	paSelectExp1: 1494, 1620.
paInfinitiveExp: 1494, 1541, 1547.	paSemicolonExp: 1494, 1499.
paIsExp: 1494, 1608.	PassTime: <u>83</u> , 85.
paLeftCurledExp: 1494, 1655.	paStExp: 1494, 1537.
paLeftDoubleExp1: 1494, 1510.	paStillNotImplemented: 1494, 1649.
paLeftDoubleExp3: 1494, 1619.	paStructExp1: 1494, 1617.
paLeftParenthExp: 1494, 1581.	paSuchExp: 1494, 1567.
paLeftSquareExp: 1494, 1582.	paSupposeOrCaseExp: 1494, 1594, 1597.
paMeansExp: 1494, 1582.	paThatExp1: 1494, 1568.
Panic mode: 1492.	paThatExp2: 1494, 1567.
paNotExpected: 1494, 1542, 1547.	paToExp: 1494, 1647.
paNumExp: 1494, 1573, 1575.	Pattern Kind Lookup Table: 1101, 1105.
paOfExp: 1494, 1516, 1649.	PatternObj: 933, 934, 935, 937, 939, 941.
paPerExp: 1494, 1593.	PatternPtr: 933, 968, 969, 997, 998, 1014, 1061,
paProofExp: 1494, 1598, 1657.	1097, 1129, 1143, 1148, 1185, 1248.
ParamCount: 48, 49, 50, 52, 67, 68, 69, 70,	paTypeExpInAdjectiveCluster: 1494.
87, 477, 478.	paTypeOrAttrExp: 1494, 1548.
ParamNr: 48.	paTypeUnexpInClusterRegistration: 1494.
ParamStr: 48, 49, 50, 52, 67, 68, 69, 70, 477,	paUnexpAntonym1: 1494.
478, 482.	paUnexpAntonym2: 1494.
paRightBraExp1: 1494, 1511, 1514.	paUnexpConnective: 1494, 1556.
paRightBraExp2: 1494, 1611.	paUnexpEquals: 1494, 1628.
paRightCurledExp1: 1494, 1513.	paUnexpHereby: 1494.
paRightCurledExp2: 1494.	paUnexpItemBeg: 1494, 1633.
paRightCurledExp3: 1494, 1654.	paUnexpOf: 1494, 1526.
paRightDoubleExp1: 1494, 1510.	paUnexpOy: 1494, 1527.
paRightDoubleExp2: 1494, 1619.	paUnexpReconsider: 1494.
- •	pa $UnexpReconstaer$: 1494. pa $UnexpRedef$: 1494, 1636.
paRightParenthExp1: 1494, 1522, 1529.	pa o nexp neaej: 1494, 1000.

paUnexpSynonym: 1494.940, 941, 942, 954, 955, 974, 975, 976, 977, paUnpairedSymbol: 1494, 1499. 987, 988, 989, 990, 991, 992, 993, 994, 997, paWithExp: 1494, 1648.998, 1014, 1018, 1020, 1058, 1060, 1071, 1097, 1107, 1109, 1113, 1115, 1117, 1121, 1122, 1124,paWrongAfterThe: 1494, 1515. 1127, 1128, 1130, 1134, 1135, 1141, 1143, 1148, paWrongAttrArgumentSuffix: 1494. 1156, 1160, 1161, 1169, 1176, 1181, 1184, 1199, paWrongAttrPrefixExpr: 1494. paWrongFormulaBeq: 1494, 1549. 1228, 1234, 1235, 1238, 1241, 1242, 1247, 1248, 1256, 1260, 1269, 1271, 1274, 1278, 1279, 1283, paWrongFunctorPatternBeq: 1494, 1609. 1286, 1291, 1294, 1296, 1303, 1317, 1324, 1329, paWrongItemBeg: 1494, 1580, 1633. 1331, 1335, 1343, 1346, 1355, 1356, 1359, 1381, paWrongJustificationBeg: 1494, 1576. paWrongModePatternBeg: 1494, 1604. 1393, 1398, 1411, 1426, 1427, 1436, 1437, 1439, 1462, 1463, 1465, 1466, 1467, 1485. paWrongModePatternSet: 1494, 1605. Plist: 878, 893, 894, 895, 896, 1109, 1110, 1130, paWrongPattBeq1: 1494, 1631. paWrongPattBeg2: 1494, 1632. 1134, 1332, 1344, 1347. PObject: 186, 188, 189, 190, 195, 201, 206, 211, paWrongPattBeg3: 1494.paWrongPredPattern: 1494, 1612. 218, 219, 231, 245, 249, 253, 266, 278, 281, 285, 291, 309, 315, 323, 326, 329, 330, 331, 333, 860, paWrongPredSymbol: 1494, 1544. 948, 1099, 1130, 1298, 1302, 1411, 1466. paWronqRadTypeBeq: 1494, 1525. paWrongReferenceBeg: 1494, 1572, 1574. Pobject: 949.paWrongRightBracket1: 1494.Pointer: 194. paWrongRightBracket2: 1494.Pointer: 184, 194, 195, 201, 202, 203, 204, 205, paWrongSchemeVarQual: 1494, 1655. 207, 209, 211, 215, 219, 222, 223, 224, 225, 226, paWrongScopeBeq: 1494, 1538. 227, 235, 238, 242, 243, 250, 254, 255, 256, 257, paWrongTermBeg: 1494, 1520, 1531. 260, 263, 264, 267, 270, 271, 272, 273, 274, 276, PByteArray: 185. 277, 282, 286, 288, 289, 290, 291, 292, 293, 295, PCharSet: 185.297, 298, 390, 461, 469, 470, 647, 659. PCMizarReleaseNbr: 77, 80, 481. pointer: 86, 89, 223, 295, 298, 299, 301, 302, PCMizarVariantNbr:77, 80, 481. 304, 486, 494, 524, 632. PCMizarVersionNbr: 77, 80, 481. Pop: 686, 687, 693, 695, 703, 704, 1205, 1213, PCMizarVersionStr: 71, 78, 82. 1225, 1249. pcmizver: 55, 77, 83, 463, 514. Pos: 85, 103, 105, 106, 107, 479, 485, 558, 560, PCollection: 219. 561, 564, 614, 640, 641. PFileDescr: 466, 470, 471, 472, 473, 474. Position: 85, 103, 105, 106, 107, 485, 494, 533, PFileDescrCollection: 469. 551, 552, 601, 602, 604, 612, 723, 738, 739, 741, $PIntItem: \underline{299}, 301.$ 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, PIntKeyCollection: 299. 752, 753, 756, 757, 758, 759, 760, 761, 762, 763, PIntSequence: 402. 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, PIntSet: 418. 774, 775, 776, 777, 778, 779, 780, 781, 782, 783, pIntSet: 418. 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, PItemList: 194, 195, 215, 243, 256, 257, 277. 794, 795, 797, 798, 799, 800, 801, 802, 803, 804, pl: 1426, 1428. 805, 806, 807, 808, 810, 811, 812, 813, 814, 815, PlaceHolderName: 722, 1046. 816, 817, 818, 819, 820, 821, 822, 823, 824, 825, PlaceholderTermObj: 758, 759. 826, 827, 828, 829, 830, 831, 832, 833, 834, 835, PlaceholderTermPtr: 758, 1046, 1121, 1168, 1454. 836, 837, 838, 839, 840, 841, 842, 843, 844, 845, PlatformNameStr: 71, 78, 81. 846, 847, 848, 849, 856, 857, 858, 859, 860, 861, PList: 195, 201, 745, 746, 751, 752, 762, 763, 764, 864, 865, 867, 868, 869, 871, 872, 875, 876, 877, 765, 766, 767, 768, 769, 776, 777, 782, 783, 784, 878, 879, 880, 881, 882, 884, 885, 886, 887, 888, 785, 798, 799, 800, 801, 802, 803, 804, 805, 810, 889, 915, 917, 924, 925, 926, 927, 928, 929, 930, 811, 812, 813, 814, 815, 816, 817, 818, 819, 856, 931, 933, 934, 935, 936, 937, 938, 939, 940, 859, 877, 884, 885, 886, 887, 888, 889, 890, 941, 942, 946, 947, 950, 951, 954, 955, 956, 891, 893, 895, 900, 901, 906, 907, 916, 918, 957, 958, 959, 960, 961, 962, 963, 964, 965, 926, 927, 930, 931, 935, 936, 937, 938, 939, 966, 967, 968, 969, 970, 971, 972, 973, 974,

975, 976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000, 1001, 1002, 1003, 1004, 1097, 1100, 1108, 1110, 1111, 1112, 1113, 1114, 1115, 1118, 1119, 1120, 1121, 1123, 1125, 1126, 1127, 1128, 1130, 1131, 1132, 1133, 1134, 1136, 1138, 1139, 1140, 1141, 1143, 1144, 1145, 1146, 1205, 1225, 1241, 1245, 1248, 1252, 1258, 1262, 1287, 1297, 1331, 1338, 1346, 1348, 1382, 1387, 1388, 1417, 1496, 1498, 1499, 1577, 1591, 1598, 1634, 1638, 1650, 1654. $post_qualified_type$: 1507. Pragma: 614. Pragma: 724, 1585, 1635, 1639, 1650, 1661. Pragma, ::\$E0F: 614. PragmaObj: 862, 863. PragmaPtr: 862, 1079, 1142, 1191, 1215. pragmas: 1193, 1199, 1492. pred: 206, 209, 221, 228, 252. Predicate: 1544. Predicate Definition Obj: 964, 965.PredicateDefinitionPtr: 964, 1089, 1143, 1191, 1267. PredicateName: 1007, 1009, 1010, 1011, 1035, 1036, 1062, 1165, 1185. PredicatePatternObj: 939, 940.PredicatePatternPtr: 939, 964, 965, 1062, 1097, 1128, 1143, 1185, 1267, 1319. PredicateSegment: 884, 1081, 1143, 1191, 1343. PredicateSegmentPtr: 886, 1081, 1143, 1191. PredicateSymbol: 724, 1460, 1461, 1540, 1541, 1544, 1546, 1547, 1549, 1580, 1612, 1630, 1631, 1632. PredicativeFormulaObj: 812, 813. PredicativeFormulaPtr: 812, 1035, 1117, 1165, PredMaxArgs: 723, 727, 728, 1539, 1543, 1613. PrefixFormatChars: 666. Prel directory: 472. prephan: 128. Previous: 307, 694, 696, 704, 714, 715, 1211. PrevPos: 723, 726, 1249, 1263, 1285, 1347, 1352, 1361, 1362, 1377, 1379, 1498, 1636, 1654. *PrevWord*: 723, 726. .prf File: 727. Primitive Recursive Function: 922. *Print_Adjective*: <u>1148</u>, 1155, 1156. Print_AdjectiveList: 1148, 1156, 1162, 1165, 1191. $print_arguments$: 1021. Print_AssumptionConditions: 1148, 1182, 1191. Print_BinaryFormula: 1148, 1163, 1165. Print_Block: 1148, 1174, 1180, 1187, 1191.

Print_Char: 1148, 1150, 1152, 1153, 1154, 1168. $Print_CompactStatement: 1148, 1172, 1174.$ Print_Conditions: <u>1148</u>, 1181, 1182, 1191. Print_Definiens: <u>1148</u>, 1186, 1191. Print_Formula: 1148, 1163, 1165, 1168, 1171, 1186, 1191. Print_Implicitly Qualified Variable: 1148, 1158, 1159. Print_Indent: <u>1148</u>, 1154, 1187, 1191. Print_Item: <u>1148</u>, 1187, 1188, 1191. Print_Justification: <u>1148</u>, 1172, 1174, 1180, 1191. Print_Label: 1148, 1170, 1171, 1174, 1186, 1191. Print_Linkage: 1148, 1173, 1174, 1191. Print_Loci: 1148, 1184, 1185, 1191. Print_Locus: <u>1148</u>, 1183, 1184, 1185, 1191. Print_NewLine: 1148, 1151, 1174, 1181, 1186, 1187, 1191. $Print_Newline: 1191.$ Print_Number: <u>1148</u>, 1152, 1168, 1176, 1179. Print_OpenTermList: 1148, 1160, 1162, 1164, 1165, 1167, 1168. Print_Pattern: <u>1148</u>, 1185, 1191. Print_PrivateFunctorTerm: 1148, 1167, 1168. Print_PrivatePredicativeFormula: 1148, 1164, 1165. Print_Proposition: <u>1148</u>, 1171, 1172, 1181, 1182, 1191. *Print_Reference*: <u>1148</u>, 1175, 1176. Print_References: <u>1148</u>, 1176, 1177, 1179. $Print_RegularStatement: 1148, 1174, 1191.$ *Print_ReservedType*: 1148, 1189, 1191. Print_SchemeJustification: 1148, 1179, 1180. Print_SchemeNameInJustification: 1148, 1178, 1179.Print_SchemeNameInSchemeHead: 1148, 1190, 1191. Print_SimpleTermTerm: 1148, 1166, 1168. Print_StraightforwardJustification: 1148, 1177, 1180. Print_String: <u>1148</u>, 1153, 1155, 1157, 1159, 1160, 1161, 1162, 1163, 1164, 1165, 1166, 1167, 1168, 1169, 1170, 1173, 1174, 1175, 1176, 1177, 1178, 1179, 1181, 1183, 1184, 1185, 1186, 1187, 1190, 1191. Print_Term: <u>1148</u>, 1160, 1161, 1165, 1168, 1174, 1186, 1191. Print_TermList: 1148, 1155, 1161, 1168. Print_TextProper: <u>1148</u>, 1188, 1192. Print_Type: 1148, 1159, 1162, 1165, 1168, 1169, 1189, 1191. Print_TypeList: <u>1148</u>, 1169, 1191.

Print_Variable: 1148, 1157, 1158, 1159, 1168, ProcessFixedVariable: 706, 708, 1289, 1566. 1191. ProcessFlexConjunction: 716, 718, 1475, 1553. Print_VariableSegment: 1148, 1159, 1165, 1191. *ProcessFlexDisjunction*: <u>716</u>, 718, 1474, 1555. Print_WSMizArticle: 1192. ProcessFuncIdentify: <u>706</u>, 708, 1283, 1648. Prior: 562, 563, 564, 566, 567, 568, 587. ProcessFuncReduction: 706, 708, 1285, 1647. private: 494.ProcessFuncSynonym: <u>706</u>, 708, 1277, 1631. PrivateFunctorDefinitionObj: 893, 894. ProcessFunctorSymbol: <u>706</u>, 708, <u>716</u>, 718, <u>1322</u>, PrivateFunctorDefinitionPtr: 893, 1084, 1143, 1429, 1519, 1610. 1191, 1355. ProcessHolds: 717, 718, 1538. PrivateFunctorTermObj: 768, 769.ProcessHypotheses: 1563, 1564, 1568.PrivateFunctorTermPtr: 768, 1014, 1043, 1045, ProcessingEnding: 84, 88. 1097, 1119, 1148, 1167, 1168, 1441. ProcessIterativeStep: 707, 708, 1601. PrivateItem: <u>1580</u>, 1587, 1590, 1633, 1637, ProcessItTerm: 716, 718, 1453, 1508. 1650, 1660. ProcessLab: 1559, 1563, 1564, 1567, 1599, 1652, PrivatePredicateDefinitionObj: 895, 896. 1653, 1656. PrivatePredicateDefinitionPtr: 895, 1084, 1143, ProcessLabel: 706, 708, 1365, 1559, 1562. 1191, 1356. ProcessLeftLocus: 706, 708, 1285, 1648. PrivatePredicativeFormulaObj: 818, 819. ProcessLeftParenthesis: 717, 718, 1503, 1519, PrivatePredicativeFormulaPtr: 818, 1014, 1026, 1528. 1027, 1097, 1114, 1148, 1164, 1165, 1469. ProcessLink: 693, 697, 1205, 1218, 1580, 1589, Procedure: 1492.1590. $process_notation_item: 1277.$ ProcessLocusTerm: 716, 718, 1454, 1508. $process_postqualified_variables: 1507.$ ProcessMeans: 706, 708, 1299, 1626. ProcessArguments: 1520, 1524, 1642. ProcessModePattern: 707, 708, 1606. ProcessAtomicFormula: 716, 718, 1459, 1544, ProcessModeSymbol: <u>716</u>, 718, 1408, 1525. 1548. ProcessModeSynonym: 706, 708, 1277, 1631. ProcessAttrAntonym: 706, 708, 1277, 1632. ProcessNegation: 716, 718, 1472, 1548. ProcessAttribute: 716, 718, 1395, 1524, 1641. ProcessNegative: 716, 718, 1471, 1541, 1547, 1549. ProcessAttributeName: 502, 509, 513. ProcessNon: 716, 718, 1392, 1524, 1641. ProcessAttributePattern: 707, 708, 1608. ProcessNot: 717, 718, 1549. ProcessAttributes: 1523, 1524.ProcessNumeralTerm: <u>716</u>, 718, 1452, 1508. ProcessAttributeValue: 502, 509, 513. ProcessPostqualification: 1506, 1513, 1517. ProcessAttrIdentify: 707, 708.ProcessPostqualifiedVariable: 716, 718, 1434, ProcessAttrSynonym: 706, 708, 1277, 1631. 1507. ProcessBegin: 693, 697, 1205, 1214, 1659, 1661. ProcessPragma: 693, 697, 1205, 1215, 1585. ProcessBeing: 706, 708, 1290, 1566. ProcessPragmas: <u>1585</u>, 1586, 1635, 1639, 1650, *ProcessBinaryConnective*: 716, 718, 1473, 1552, 1661. 1554, 1556. ProcessPredAntonym: 706, 708, 1277, 1632. ProcessCase: <u>1592</u>, 1594, 1596, 1597. ProcessPredicateSymbol: <u>706</u>, 708, <u>716</u>, 718, 1318, Process Choice: 1567, 1569, 1580. 1460, 1543, 1612. ProcessComment: 632, 640, 641, 642. ProcessPredIdentify: 707, 708. Process Contradiction: 716, 718, 1470, 1549. ProcessPredSynonym: 706, 708, 1277, 1631. ProcessCorrectness: <u>706</u>, 708, 1363, 1622. ProcessPrivateReference: 706, 708, 1373, 1572. ProcessDef: 706, 708, 1375, 1573. ProcessQua: 716, 718, 1401, 1504. ProcessDefiniensLabel: 706, 708, 1368, 1626, 1628. ProcessReconsideredVariable: 706, 708, 1357, ProcessDoesNot: 717, 718, 1542, 1547. ProcessEndTag: 502, 508, 512. 1584. ProcessRedefine: 693, 697, 1205, 1217, 1635. ProcessEquals: <u>706</u>, 708, 1299, 1628. $ProcessReservedIdentifier: \underline{706}, 708, 1347, 1651.$ ProcessExactly: 716, 718, 1403, 1504. ProcessRightLocus: <u>706</u>, 708, 1285, 1648. ProcessExemplifyingVariable: 706, 708, 1361, 1588. ProcessRightParenthesis: 717, 718, 1503, 1519,

1521, 1522, 1529, 1532, 1545, 1551.

ProcessField: 706, 708, 1333, 1620.

ProcessRightSideOfPredicateSymbol: 716, 718, QualifiedSegmentObj: 741, 742, 743, 745. 1461, 1539. QualifiedSegmentPtr: 741, 838, 839, 840, 841, 842, ProcessSch: 707, 708, 1575. 843, 1014, 1025, 1084, 1097, 1113, 1115, 1148, $ProcessSchemeName: \quad \underline{706}, \ 708, \ 1335, \ 1654.$ 1159, 1168, 1191, 1263, 1487, 1488. ProcessSchemeNumber: <u>706</u>, 708, 1377, 1575. Qualified TermObj: 786, 787. ProcessSchemeReference: 706, 708, 1369, 1574. Qualified TermPtr: 786, 1055, 1121, 1168, 1402. ProcessScheme Variable: 706, 708, 1342, 1655. QualifyingFormulaObj: 820, 821. ProcessSentence: <u>1560</u>, 1563, 1564, 1567, 1599, QualifyingFormulaPtr: 820, 1039, 1115, 1165, 1652, 1653, 1656. 1466. ProcessSimpleTerm: 716, 718, 1400, 1509. QuantifiedFormulaObj: 838, 839, 840, 842.ProcessStructureSymbol: 706, 708, 1327, 1617. QuantifiedFormulaPtr: 838, 1040, 1041, 1165. ProcessThe: <u>717</u>, 718, 1515. Quantified Variables: <u>1535</u>, 1537, 1538. Process Theorem Number: 706, 708, 1376, 1573. QuickSort: 309, 334, 337. Process Thesis: 716, 718, 1458, 1549. Quicksort: 257, 334, 337, 403. Process Variable: 716, 718, 1486, 1536. QuietMode: 64, 67, 68, 69, 73, 85. QuoteStrForXML: 514, 516, 553, 587, 1010, Process Visible: 706, 708, 1325, 1602. 1046. 1079. program: 5. Prohibited: 633, 638. QuoteXMLAttr: 514, 519. ProofPragma: 1193, 1195, 1219, 1378, 1578, r: 132, 151, 154, 155, 156, 157, 159, 160. 1600, 1657. Radix type: 1525. PropertiesProcessing: 61, 65, 66. RadixTypeObj: 798, 799, 800, 802. RadixTypePtr: 798. PropertyKind: 720, 722, 984, 985, 999, 1000, 1001, 1002, 1101, 1106, 1143, 1232, 1233, 1242, 1649. RadixTypeSubexpression: 1525, 1530, 1548, 1644, PropertyKindLookupTable: 1101, 1102, 1106. 1645, 1646. PropertyName: 722, 1079, 1101, 1191. Rational: 130, 131, 132, 134, 160, 161, 162, 163, PropertyObj: 984, 985. 164, 165, 166, 167, 168, 177, 179. PropertyPtr: 984, 1079, 1087, 1143, 1191, 1250. RationalAdd: 132, 161, 173, 176, 177, 179. PropertyRegistrationObj: 999, 1000, 1001.RationalDiv: 132, 166, 177. PropertyRegistrationPtr: 999, 1094, 1191. Rational Eq: 132, 167, 170.PropositionObj: 864, 865. Rational GEQ: 18.PropositionPtr: 864, 913, 914, 916, 918, 972, 973, RationalGT: 18, 132, 168, 171. 1014, 1069, 1097, 1132, 1137, 1148, 1171, 1219, RationalInv: 132, 165, 166. 1250, 1259, 1261, 1345, 1359, 1360. RationalLE: 132, 168, 171. Prune: 219, 232. RationalLT: 18. PSortedCollection: 286.RationalMult: 132, 164, 166, 176, 177, 179. PStr: <u>191</u>, 282. RationalNeg: 132, 163, 175. PString: 185, 297, 309, 338, 339, 461, 466. RationalReduce: 132, 160, 161, 162, 164. PStringCollection: 295.RationalSub: 132, 162, 174, 176, 177. PStringItem: 309.RComplex: 130, 131, 133, 134, 135, 169, 170, 171, PStringItemList: 309, 335.172, 173, 174, 175, 176, 177, 178, 179, 181. PStringList: 309. Re: 130, 131, 169, 170, 171, 172, 173, 174, 175, PSymbol: 562, 576, 577, 578, 580, 587. 176, 177, 179, 181. PublicLibr: 63, 70. Read: 579, 727. PUsortedStringCollection: 295. Read_Adjective: 1097, 1108, 1109. PutError: 86, 87, 103, 106, 108. Read_AdjectiveList: 1097, 1109, 1110, 1115, 1143. PutObject: 309, 315, 330, 333. Read_AttributePattern: 1097, 1126, 1129, 1143. PutString: 309, 332. Read_Block: 1097, 1145, 1146. Read_CompactStatement: 1097, 1137, 1141, 1143. PVocabulary: 557, 573, 581, 582, 583, 584, 585. PWordArray: 185.Read_Definiens: 1097, 1130, 1143. *P2*: 17. Read_Formula: 1097, 1115, 1116, 1117, 1121, *P3*: **17**. 1130, 1132, 1143.

Read_FunctorPattern: 1097, 1127, 1129, 1143.

QT: 489, 500, 507, 509.

Read_ImplicitlyQualifiedSegment: 1097, 1112, ReadTypeList: 1579, 1581, 1582, 1655. 1113. ReadVisible: 1603, 1606, 1607, 1608, 1611, 1612, 1613, 1617. Read_InternalSelectorTerm: 1097, 1120, 1121. Read_Item: 1097, 1144, 1145, 1146. ReadWord: 1500, 1503, 1504, 1505, 1507, 1508, $Read_ItemContents: 1097, 1143, 1146.$ 1509, 1510, 1511, 1512, 1514, 1515, 1516, 1517, $Read_ItemContentsAttr: 1097, 1142, 1146.$ 1519, 1521, 1524, 1525, 1526, 1527, 1528, 1535, Read_Justification: 1097, 1137, 1140, 1141, 1143. 1538, 1539, 1542, 1543, 1545, 1547, 1548, 1549, $Read_label$: 1132. 1550, 1552, 1553, 1554, 1555, 1556, 1559, 1564, 1568, 1569, 1570, 1571, 1572, 1573, 1574, 1575, Read_Label: 1097, 1130, 1131, 1141. 1580, 1581, 1582, 1583, 1584, 1587, 1588, 1589, $Read_LocalReference: 1097, 1133, 1134.$ 1590, 1596, 1601, 1605, 1606, 1611, 1613, 1615, Read_Loci: 1097, 1124, 1125, 1126, 1127, 1128, 1616, 1621, 1622, 1623, 1624, 1626, 1627, 1628, 1143. 1629, 1631, 1632, 1633, 1634, 1637, 1638, 1640, Read_Locus: 1097, 1123, 1124, 1126, 1143. 1641, 1643, 1644, 1645, 1646, 1647, 1648, 1649, Read_ModePattern: <u>1097</u>, 1125, 1129, 1143. 1650, 1651, 1652, 1653, 1654, 1655, 1656, 1657. Read_Pattern: <u>1097</u>, 1129, 1143. Reasoning: 1577, 1578, 1587, 1598, 1599, 1600. Read_PredicatePattern: 1097, 1128, 1129, 1143. ReasPos: 1577, 1598. Read_PrivateFunctorTerm: 1097, 1119, 1121. Reconsider (statement): 899. Read_PrivatePredicativeFormula: 1097, 1114, reconsider: 899. 1117. record: 6. Read_Proposition: <u>1097</u>, 1132, 1137, 1143. Read_References: 1097, 1134, 1138, 1139. $record_separator$: <u>610</u>, 617, 637. Reduce Registration Obj: 1003, 1004. $Read_RegularStatement: 1097, 1141, 1143.$ ReduceRegistrationPtr: 1003, 1094, 1143, 1191, Read_ReservationSegment: 1097, 1135, 1143. Read_SchemeJustification: 1097, 1139, 1140. 1275.Reduction: $\underline{1647}$, $\underline{1650}$. Read_SchemeNameInSchemeHead: 1097, 1136, ReductionProcessing: 61, 65, 66. 1143. ReferenceKind: 867.Read_SimpleTerm: 1097, 1118, 1121. ReferenceKindName: 1070, 1071, 1134. Read_StraightforwardJustification: 1097, 1138, 1140. ReferenceObj: 867, 868, 870. Reference Ptr:~867,~1071,~1176,~1376.Read_Term: <u>1097</u>, 1107, 1115, 1121, 1130, 1141, ReferenceSort: 724, 1375, 1573, 1575. 1143. Refuted: 379, 382, 386, 387, 388. Read_TermList: <u>1097</u>, 1107, 1108, 1110, 1114, 1117, 1119, 1121. RegisterCluster: 1643, 1650. $Read_TextProper: \underline{1097}, 1144, 1147.$ Register Property: 1649, 1650.RegistrationBlock: 1650, 1660. Read_Type: 1097, 1110, 1113, 1115, 1121, 1122, Regular Statement: 1580, 1589, $\underline{1599}$. 1135, 1143. Read_TypeList: 1097, 1122, 1143. RegularStatementKind: 909, 910, 911, 912, 1225. Read_Variable: 1097, 1111, 1112, 1113, 1135, 1143. RegularStatementName: 1079, 1141. Read_VariableSegment: <u>1097</u>, 1113, 1115, 1121, RegularStatementObj: 909, 910, 911, 913. RegularStatementPtr: 909, 1014, 1076, 1079, 1080, 1143. $Read_WSMizArticle$: 1147. 1097, 1141, 1148, 1174, 1191. readln: 577, 582, 597, 636, 1009, 1012. RenameFile: 34, 36.ReadLn: 472, 475, 476, 477, 478, 479, 499, $report_mismatch$: 505. 568, 578, 583. ReportTime: 85, 86, 116, 126. Repr: 562, 563, 564, 565, 566, 567, 568, 570, 587. ReadNames: $\underline{464}$, $\underline{476}$. ReadParams: 1607, 1610.Reprs: 573, 574, 575, 577, 578, 580, 587. ReadPrivate Voc: 573, 576, 581. $req_info: 128.$ ReadSortedNames: 464, 475.*Reservation*: 1651, 1660. ReadToken: 723, 726, 1492. Reservation Segment Obj: 890, 891.ReadTokenProc: 1492, 1496, 1499, 1500, 1577, ReservationSegmentPtr: 890, 1014, 1077, 1080, 1585, 1598, 1658, 1661, 1662. 1097, 1135, 1191, 1347. ReadTokenProcedure: 1492.Reset: 46, 472, 475, 476, 477, 478, 480, 496, 583.

reset: 576, 582, 596, 645, 727, 1009, 1012. SchemePtr: 888, 1014, 1078, 1081, 1097, 1136, 1143, 1148, 1190, 1191, 1250. ResetAccOptions: 66, 67.result: 858, 1005, 1006, 1099, 1100, 1107, 1108, SchemeSegmentKind: 884, 885. 1109, 1110, 1111, 1112, 1113, 1115, 1116, 1117, SchemeSegmentName: 1081, 1143. 1118, 1119, 1120, 1121, 1122, 1123, 1124, 1125, SchemeSegmentObj: 884, 885, 886. 1126, 1127, 1128, 1129, 1130, 1131, 1132, 1133, SchemeSegmentPtr: 884, 886, 1081, 1191, 1343. 1134, 1135, 1136, 1137, 1138, 1139, 1140, 1141, SchemesProcessing: 61, 65, 66. 1144, 1145, 1146, 1147, 1201. $screaming_run_on_case$: 22. Result: 460, 516, 517, 518, 519, 1005, 1006, 1114, scTooLongLineErrorNr: 722, 726, 1496. Search: 18, 250, 263, 286, 289, 290, 291, 293, 1115, 1117, 1121. Rewrite: 108, 109, 473, 486, 534, 584. 305, 350, 352, 354, 356, 357, 363, 429, 433, rewrite: 599, 1149. 437, 439, 440, 442, 443, 444, 445, 447, 451, RightBracketName: 1007, 1009, 1010, 1011, 1048, 455, 457, 458, 460, 595, 624. 1064, 1168, 1185. SearchPair: 359, 363, 365, 372, 373, 377, 378, RightCircumfixSymbol: 724, 1511, 1514, 1611. 382, 383, 384, 385, 387. RightSideOfPredicativeFormulaObj: 810, 811, 812. Sedgewick, Robert: 258. RightSideOfPredicativeFormulaPtr: 810, 1036, seekEOF: 576.1117, 1165, 1463. seekEof: 475, 476, 477, 478, 479, 597, 1009, 1012. Ritchie, Dennis M.: 922. SegmentKind: 741, 742. ROne: 131.SegmentKindName: 1024, 1025, 1113, 1121, 1143. RoundedBracket: 722, 1011. SelectorName: 1007, 1009, 1010, 1044, 1050, raRealAdd: 173.1091, 1168, 1191. rgRealDiff: 174.Selector Symbol: 724, 1515, 1620, 1630.rqRealDiv: 177.SelectorTermObj: 772, 773.rqRealMult: 176.SelectorTermPtr: 772, 1050, 1121, 1168, 1447. rqRealNeg: 175.Self: 187, 188, 189, 201, 217, 335, 367, 370, 376, rs: 149, 150. 380, 389, 427, 678. RTErrorCode: 90, 91, 103, 112, 113, 114. SemErr: 1253, 1409, 1443, 1492, 1501. Rudnicki, Piotr: 920. Semicolon: 1499, 1586, 1592, 1593, 1622, 1623, RunError: 112, 113, 114, 183, 208, 321, 342, 394, 1635, 1640, 1643, 1647, 1648, 1649, 1650, 412, 430, 448, 521, 525. 1657, 1660. RunTimeError: 95, 99, 105, 107, 114, 493, 677, sErrProperty: 720, 722, 1233. 684, 685, 686, 687, 1478, 1508. SetAttributeValue: 502, 513. RuntimeError: 678. $SetCapacity: 309, 310, 311, 317, 324, 335, \underline{402},$ RZero: 131, 165. 404, 407, 408, 410, 415, 417, 419, 429, 431, 432, r1: 132, 161, 162, 163, 164, 166, 167, 168. 433, 435, 447, 449, 450, 451, 453. r2: 132, 161, 162, 164, 166, 167, 168.SetConsoleCtrlHandler: 97, 98. $s: \frac{79}{170}$. SetConsoleMode: 98. $SaveDct: \underline{593}, \underline{599}.$ SethoodRegistrationObj: 1001, 1002.SaveXDct: <u>593</u>, 600, 733. SethoodRegistrationPtr: 1001, 1094, 1143, 1191, scanner: 588, 647, 719, 850. 1275, 1312. SchemeBlock: 1654, 1660. setlength: 453, 1009, 1012, 1208, 1390, 1404, SchemeJustificationObj: 881, 882. 1421, 1429. SchemeJustificationPtr: 881, 1014, 1073, 1074, SetLimit: 195, 197, 199, 200, 204, 205, 215, 216, 1097, 1139, 1148, 1178, 1179, 1180, 1369, 217, 220, 223, 232, 235, 236, 238, 242, 243, 250, 251, 253, 254, 255, 256, 267, 276, 277, 341, 343, 1377, 1379, 1380. SchemeListsProcessing: 61, 65, 66, 67. 344, 345, 347, 352, 353, 360, 362, 363, 388. SchemeObj: 888, 889. SetParserPragma: 1193, 1195, 1585. SchemePos: 1654, 1657. SetSorted: 309, 336, 337. SchemePragmaIntervals: 1193, 1197, 1198. SetStringLength: 24, 26. SchemePragmaOff: 1193, 1196, 1197, 1198. SetSym: 722, 1011. SchemePragmaOn: <u>1193</u>, 1196, 1197, 1198. settextbuf: 645.

SetTextBuf: 496, 526.StandardPriority: 555, 564, 567. ShortString: 185.StandardTypeObj: 800, 801.shr: 7. Standard TypePtr: 800, 1021, 1110, 1162, 1409. SIGINT: 95.Start: 1387, 1390, 1427, 1429, 1430, 1459. Signal Handler: 95.StartAdjectiveCluster: 717, 718, 1523. SignatureProcessing: 61, 65, 66, 67. StartAggregateTerm: 716, 718, 1444, 1510. StartAggrPattSegment: 706, 708, 1332, 1620. SIGQUIT: 95. SIGTERM: 95.StartArgument: <u>717</u>, 718, 1533. SimpleDefiniens: 946, 951, 1068, 1130, 1186. StartAssumption: 706, 708, 1294, 1568. Simple Definiens Obj: 950, 951.StartAtomicFormula: 717, 718, 1549. SimpleDefiniensPtr: 950, 1068, 1130, 1186, 1385. StartAtSignProof: 693, 697, 1577. SimpleFraenkelTermObj: 782, 783, 784. StartAttributeArguments: 716, 718, 1396, 1524, SimpleFraenkelTermPtr: 782, 1054, 1121, 1168, StartAttributePattern: 706, 708, 1310, 1608. 1439. SimpleJustification: <u>1576</u>, 1578, 1580, 1584, 1593, StartAttributes: 706, 708, 716, 718, 1279, 1391, 1599, 1601, 1633. 1530, 1548, 1641, 1643, 1644, 1645, 1646. SimpleJustificationObj: 877, 878, 879, 881. StartAttrIdentify: 707, 708. StartBracketedTerm: 716, 718, 1442, 1505, 1512. SimpleJustificationPtr: 877, 900, 901, 906, 907, 915, 917, 1143, 1255, 1257, 1373, 1376. StartChoiceTerm: 716, 718, 1450, 1515, 1516, Simple TermObj: 756, 757. SimpleTermPtr: 756, 1014, 1042, 1045, 1097, StartCollectiveAssumption: 706, 708, 1296, 1564. 1118, 1148, 1166, 1168, 1400. StartCondition: 707, 708, 1567.Single Assumption: 970, 973, 1092, 1143, 1182. StartConstructionType: 706, 708, 1364, 1621. Single Assumption Obj: 972, 973.StartDefiniens: 706, 708, 1301, 1626. StartDefPredicate: 707, 708, 1623, 1631, 1632. Single Assumption Ptr: 972, 1092, 1143, 1182, 1259. StartEquals: 707, 708, 1628. Singleton: 219, 231. SizeOf: 184, 187, 189, 201, 215, 223, 243, 252, StartExemplifyingTerm: 706, 708, 1362, 1588. 254, 256, 277, 279, 318, 328, 335, 340, 347, 352, StartExistential: 716, 718, 1481, 1537. 353, 355, 363, 366, 380, 388, 411, 415, 417, 419, StartExpansion: 706, 708, 1307, 1624. 433, 435, 438, 451, 456, 572, 574. StartFields: 706, 708, 1329, 1615. $skip_all_other_ids$: 503. StartFixedSegment: 706, 708, 1288, 1566. $skip_spaces$: 501. StartFixed Variables: <u>706</u>, 708, 1286, 1565. $skip_to_begin$: 1658. StartForgetfulTerm: 716, 718, 1448, 1516. $skip_to_quotes$: 501. StartFraenkelTerm: 716, 718, 1431, 1513, 1517. StartFuncIdentify: 706, 708, 1283, 1648. $skip_xml_prolog: \underline{503}.$ $skipped_proof_justification: \underline{1219}, \underline{1220}.$ StartFuncReduction: 706, 708, 1285, 1647. Skyscraper: 920. StartFunctorPattern: 706, 708, 1321, 1609. SliceIt: 604, 613, 629. StartGuard: 706, 708, 1303, 1627, 1629. Soft type: 796. StartIterativeStep: 706, 708, 1383, 1601.StartJustification: 706, 708, 1378, 1578, 1599, SOLARIS: 81. Sort: 250, 259, 265, 309, 336, 337. 1600, 1601. Source: 46. StartLibraryReferences: 706, 708, 1371, 1573. Specification: <u>1614</u>, 1620, 1655. StartLongTerm: 716, 718, 1413, 1519. Spelling: 604, 605, 606, 607, 721, 726, 729, StartModePattern: 706, 708, 1313, 1604. 1215, 1585. StartMultiPredicativeFormula: 716, 718, 1464, SplitFileName: 34, 38, 39, 40, 41, 42.1540. StartNewType: <u>707</u>, 708, 1584. SquareBracket: 722, 1011, 1321. StartOtherwise: 707, 708, 1627, 1629. StackedObj: 307, 308, 693, 703, 713. StackedPtr:307.StartPostqualification: 716, 718, 1432, 1506. $Standard Mode Definition Obj: \quad 960, \ \underline{961}.$ StartPostqualificationSpecyfication: 716, 718, StandardModeDefinitionPtr: 960, 1088, 1143, 1435, 1507. 1191, 1264. StartPostqualifyingSegment: 716, 718.

StartPostQualifyingSegment: 1433, 1507. stillcorrect: 1311. StartPredicatePattern: <u>706</u>, 708, 1315, 1612. stIterativeEquality: 909, 918, 1076, 1079, 1141, StartPredIdentify: 707, 708. 1174, 1261, 1381. StartPrefix: 707, 708, 1616. StopOnError: 64, 67, 68, 69, 86, 106. StartPrivateConstant: <u>706</u>, 708, 1353, 1583. Store: <u>562</u>, 569, 580. StartPrivateDefiniendum: 706, 708, 1349, 1581, StoreFIL: 469, 473. 1582. StoreFormats: 665, 679. StartPrivateDefiniens: 706, 708, 1354, 1581, 1582. StoreMmlVcb: 557, 584.StartPrivateFormula: <u>716</u>, 718, 1468, 1550. StoreMmlVcbX: 557, 585. StartPrivateTerm: 716, 718, 1440, 1509. Store Voc: 573, 580, 584. StartProperText: 693, 697, 1205, 1216, 1659. Str: 33, 79, 80, 125, 126, 143, 144, 145, 146, StartQualifiedSegment: 716, 718, 1483, 1535. 148, 170, 481, 567, 605. StartQualifyingType: 716, 718, 1484, 1535. StraightforwardJustificationObj: 879, 880. StartReferences: 707, 708, 1571. StraightforwardJustificationPtr: 879, 1072, 1074, StartRegularStatement: 706, 708, 1366, 1562, 1097, 1138, 1148, 1174, 1177, 1180, 1191, 1379. 1599. $StreamObj: 520, \underline{521}, \underline{522}, \underline{523}, 533.$ StartReservationSegment: 706, 708, 1347, 1651. StrictSym: 722, 1011. StartRestriction: 716, 718, 1476, 1538. string: 284, 285, 296, 312, 315, 320, 322, 325, 326, $starts_with_term_token$: 1549. 327, 328, 330, 331, 332, 334, 462, 464, 466, 467, StartScaner: 723, 729, 730, 732. 469, 472, 473, 474, 475, 476, 477, 478, 479, 485, StartScanner: 726.487, 490, 491, 493, 494, 495, 502, 503, 513, 514, StartSchemeDemonstration: 693, 700, 1205, 516, 517, 519, 520, 522, 524, 526, 528, 529, 530, 1219, 1657. 531, 532, 533, 534, 535, 541, 543, 545, 547, 548, StartSchemeLibraryReference: 706, 708, 1372, 549, 550, 551, 553, 557, 558, 562, 563, 564, 567, 1575. 568, 573, 576, 580, 581, 582, 583, 584, 585, 591, StartSchemePremise: 707, 708, 1656. 592, 593, 595, 596, 599, 600, 601, 602, 604, 605, StartSchemeQualification: 706, 708, 1337, 1655. 621, 632, 642, 645, 665, 676, 679, 721, 722, StartSchemeReference: 707, 708, 1574. 723, 725, 727, 730, 732, 855, 856, 857, 862, StartSchemeSegment: <u>706</u>, 708, 1341, 1655. 863, 1005, 1006, 1007, 1008, 1013, 1014, 1015, StartSelectorTerm: 716, 718, 1446, 1515. 1096, 1097, 1098, 1099, 1103, 1104, 1105, 1106, StartSentence: 706, 708, 1280, 1560, 1561. 1130, 1141, 1142, 1143, 1146, 1147, 1148, 1149, StartSethoodProperties: 707, 708, 1649. 1153, 1192, 1193, 1194, 1195, 1196. String: 24, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, StartSimpleFraenkelTerm: <u>716</u>, 718, 1438, 1517. 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, StartSimpleJustification: 706, 708, 1379, 1576. 49, 50, 53, 57, 59, 60, 70, 71, 72, 74, 75, 78, StartSpecification: 707, 708, 1614. 79, 80, 81, 82, 84, 85, 87, 105, 108, 109, 116, StartTheoremBody: <u>707</u>, 708, 1652, 1653. StartType: 716, 718, 1407, 1530, 1548, 1643, 125, 126, 129, 130, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 151, 1644, 1645, 1646. 152, 153, 154, 155, 156, 157, 158, 159, 160, 170, StartUniversal: 716, 718, 1482, 1538. 180, 191, 192, 281, 309, 338, 461. Start Visible: 706, 708, 1324, 1603, 1607. StringColl: 295, 298, 464, 476, 478. StAX, Java: 528. StringListError: 309, 314, 318, 319, 321, 322, stCompactStatement: 909, 914, 1076, 1079, 1141, 323, 327, 332, 333. 1174, 1261, 1366. STD_INPUT_HANDLE: 98. StrToInt: 518. StructModeMaxArgs: 723, 727, 728, 1527, 1618. stdcall: 100.stDiffuseStatement: 909, 912, 1076, 1079, 1141, StructTypeObj: 802, 803.StructTypePtr: 802, 1021, 1110, 1162, 1409. 1174, 1261, 1366. Structure, first-class: 920. $steal_from: 257.$ StillCorrect: 1276, 1314, 1319, 1323, 1380, 1443, StructureDefinitionObj: 930, 931. StructureDefinitionPtr: 930, 1091, 1143, 1191, 1449, 1492, 1496, 1500, 1501, 1586, 1594, 1596, 1597, 1622, 1623, 1633, 1635, 1639, 1270.

Structure Mode Max Args: 727.

1650, 1657, 1660.

StructureName: 1007, 1009, 1010, 1021, 1049, sy_Case: 1187, 1497, 1577, 1587, 1591, 1595, 1596, 1051, 1091, 1162, 1168, 1191. 1633, 1637, 1650, 1660. StructureSymbol: 724, 1327, 1409, 1510, 1515, sy_case : 724. 1519, 1520, 1524, 1525, 1531, 1548, 1549, 1580, sy_cases : 724. 1617, 1630, 1641, 1642. sy_Cases: 1191, 1593, 1633. Structure symbol: 1502. $sy_cluster$: 724. Str2BlockKind: 1105, 1145. $sy_Cluster: 1191, 1650.$ Str2CorrectnessKind: 1106, 1142, 1143. sy_Colon: 724, 1365, 1368, 1512, 1513, 1559, 1573, Str2FormulaKind: 1105, 1115. 1575, 1626, 1628, 1654. Str2ItemKind: 1105, 1146. $sy_Comma: 724, 1362, 1506, 1507, 1521, 1533,$ Str2PatterenKind: 1105, 1129.1535, 1536, 1545, 1565, 1566, 1571, 1573, 1579, Str2PropertyKind: 1106, 1143.1583, 1584, 1588, 1603, 1616, 1619, 1620, 1627, Str2TermKind: 1106, 1121.1629, 1630, 1648, 1651, 1655. Str2XMLAttrKind: 1104. $sy_consider$: 724. Str2XMLElemKind: 1103. sy_Consider: 1191, 1497, 1580. Sub: 129, 144, 150, 151, 154, 155, 162. sy_Contradiction: 1165, 1549, 1580. $sy_contradiction$: 724. SubexpObj: 713, 714, 715, 718, 1388. SubexpPtr: 690, 712, 713, 715. sy_Correctness: 1191, 1363, 1622. $swap_indices: \underline{257}.$ $sy_correctness$: 724. SwitchOffUnifier: 64, 69. $sy_CorrectnessCondition$: 724, 1622. sx: 145. $sy_deffunc: 724.$ $sy_Deffunc: 1497, 1580.$ $sy_according$: 724. $sy_aggregate$: 724. $sy_DefFunc: 1191.$ *sy_all*: 724. sy_define : 724. sy_All : 1168, 1517. $sy_-definition: 724.$ sy_Ampersand: 724, 1163, 1478, 1552, 1553. sy_Definition: 1187, 1497, 1577, 1587, 1595, 1635, 1639, 1650, 1660, $su_and: 724.$ sy_And: 1181, 1191, 1563, 1567, 1656. $sy_defpred$: 724. $sy_antonym$: 724. $sy_Defpred: 1497, 1580.$ $sy_Antonym$: 1191, 1640. $sy_DefPred: 1191.$ $sy_are: 724, 1507.$ sy_Do: 1540, 1544, 1546, 1547, 1549, 1580. sy_Arrow: 724, 1191, 1364, 1614, 1621, 1643, $sy_{-}do: 724.$ 1644, 1647. sy_Does: 1165, 1540, 1544, 1546, 1547, 1549, 1580. sy_As: 1191, 1584. sy_does : 724. $sy_{-}as: 724.$ sy_Dolar: 724, 1502, 1508, 1515, 1519, 1520, sy_Assume: 1191, 1587, 1637. 1531, 1549, 1580. sy_assume : 724. sy_DotEquals: 724, 1174, 1380, 1381, 1576, 1599. sy_Ellipsis: 724, 1163, 1552, 1553, 1554, 1555. sy_Attr: 1191, 1623, 1635, 1636. $sy_-attr:$ 724. sy_end : 724. sy_End : 1187, 1497, 1499, 1577, 1586, 1591, 1595, sy_axiom : 724. 1634, 1638, 1650, 1660, 1662. $sy_Axiom: 1660.$ sy_Be : 1159, 1535, 1566. $sy_environ$: 724. $sy_be: 724.$ sy_Equal: 724, 1460, 1461, 1540, 1541, 1544, 1546, $sy_begin: 724.$ 1547, 1549, 1580, 1581, 1583, 1584, 1588, sy_Begin: 726, 1191, 1497, 1577, 1587, 1595, 1635, 1612, 1613, 1630, 1648. 1639, 1650, 1658, 1659, 1660, 1661. *sy_Equals*: 1191, 1625. sy_equals : 724. $sy_being: 724.$ $sy_Being: 1535, 1566.$ sy_Error: 724, 726, 1496, 1630. sy_By: 1177, 1379, 1576. sy_Ex: 1165, 1538, 1549, 1580. $sy_by: 724.$ $sy_ex: 724.$ $sy_canceled$: 724. $sy_exactly: 724.$ sy_Canceled: 1637, 1650, 1660. sy_Exactly: 1168, 1504, 1518.

 $sy_for:$ 724. sy_now : 724. sy_For: 1165, 1191, 1538, 1549, 1580, 1631, 1632, sy_Now: 1187, 1366, 1497, 1577, 1580, 1598, 1599. 1643, 1644, 1645, 1651. sy_Of: 1162, 1168, 1185, 1191, 1408, 1447, 1515, sy_From: 1179, 1379, 1576. 1516, 1517, 1526, 1605, 1606. $sy_from: 724.$ $sy_-of: 724, 1649.$ sy_Func: 1191, 1623, 1628, 1635, 1636. $sy_{-}Or$: 1163, 1478, 1554, 1555. $sy_-func: 724.$ $sy_{-}or: 724.$ $sy_given: 724.$ sy_Otherwise: 1186, 1627, 1629. sy_Given: 1191, 1587, 1637. $sy_otherwise$: 724. sy_hence : 724. sy_Over: 1162, 1191, 1527, 1617. sy_Hence: 1218, 1589. $sy_over: 724.$ sy_Hereby: 1497, 1577, 1587, 1598, 1633, 1637, $sy_per: 724.$ 1650, 1660. sy_Per: 1191, 1497, 1587, 1590, 1591, 1592, 1593, sy_hereby : 724. 1633, 1637, 1650, 1657, 1660. sy_holds : 724. sy_Pred: 1191, 1623, 1635, 1636. sy_Holds: 1165, 1538. $sy_pred: 724.$ $sy_identify$: 724. sy_prefix : 724. $sy_Identify: 1191, 1650.$ sy_proof : 724. $sy_If: 1186, 1626, 1627, 1628, 1629.$ sy_Proof: 1187, 1191, 1378, 1497, 1577, 1578, $sy_-if: 724.$ 1580, 1598, 1599, 1657. $sy_-iff: 724.$ sy_Property: 724, 1623, 1650. sy_Iff: 1163, 1478, 1556. $sy_provided$: 724. $sy_implies$: 724. $sy_Provided$: 1191, 1656. sy_Implies: 1163, 1478, 1556. $sy_-qua: 724.$ sy_Is: 1165, 1168, 1185, 1191, 1459, 1507, 1544, $sy_Qua: 1168, 1504, 1518.$ 1608, 1624, 1630. $sy_reconsider$: 724. $sy_is: 724.$ sy_Reconsider: 1191, 1497, 1580. sy_It: 1168, 1502, 1508, 1515, 1519, 1520, 1531, $sy_redefine$: 724. 1549, 1580. sy_Redefine: 1191, 1217, 1635. $sy_{-}it: 724.$ sy_Reduce : 1191, 1650. sy_LeftCurlyBracket: 724, 1321, 1502, 1512, 1515, sy_reduce : 724. 1519, 1520, 1531, 1549, 1580, 1609, 1630, 1655. $sy_registration$: 724. sy_LeftParanthesis: 724, 1343, 1502, 1503, 1509, sy_Registration: 1187, 1497, 1577, 1587, 1595, 1515, 1519, 1524, 1528, 1548, 1574, 1580, 1581, 1635, 1639, 1650, 1660. 1607, 1608, 1609, 1616, 1630, 1641, 1655. $sy_reserve$: 724, 1191. sy_LeftSquareBracket: 724, 1321, 1343, 1502, sy_Reserve: 1497, 1577, 1587, 1595, 1635, 1639, 1511, 1515, 1519, 1520, 1531, 1549, 1580, 1650, 1660. 1582, 1609, 1630, 1655. sy_RightCurlyBracket: 724, 1511, 1513, 1514, sy_let : 724. 1611, 1654. sy_Let: 1191, 1587, 1637, 1640, 1650. sy_RightParanthesis: 724, 1503, 1509, 1511, 1514, $sy_LibraryDirective:$ 724. 1519, 1521, 1522, 1529, 1532, 1545, 1551, 1574, $sy_means: 724.$ 1579, 1581, 1607, 1608, 1611, 1616, 1655. sy_Means: 1191, 1582, 1625. sy_RightSquareBracket: 724, 1511, 1514, 1550, sy_Mode: 1191, 1623, 1635, 1636. sy_mode : 724. 1579, 1582, 1611, 1655. sy_Scheme: 1191, 1246, 1497, 1577, 1587, 1595, sy_non : 724. 1635, 1639, 1650, 1660. sy_Non: 1155, 1392, 1515, 1524, 1548, 1641. sy_scheme : 724. sy_not : 724. sy_Not: 1165, 1542, 1547, 1548, 1549, 1580. $sy_selector$: 724. sy_Semicolon: 724, 1362, 1380, 1497, 1499, $sy_notation$: 724. sy_Notation: 1187, 1497, 1577, 1587, 1595, 1635, 1576, 1662. 1639, 1650, 1660. sy_set : 724.

sy_Set: 1168, 1191, 1408, 1515, 1524, 1525, 1548, syCorrectness: 720, 1079. 1580, 1604, 1641. syDef: 720, 1375, 1573. sy_St: 1165, 1537, 1538. syDefinitions: 720.syEqualities: 720. $sy_{-}st: 724.$ sy_Struct : 1191, 1623, 1635. syExistence: 720, 1264, 1272, 1385. sy_struct : 724. syExpansions: 720. sy_StructLeftBracket: 724, 1168, 1191, 1510, 1524, syldempotence: 720, 1233. 1619, 1641, 1642. syInvolutiveness: 720, 1233. sy_StructRightBracket: 724, 1168, 1191, 1510, syIrreflexivity: 720, 1233. 1619. SymbolCounters: 554, 571.sy_Such: 1191, 1293, 1567, 1568. SymbolIntSegArr: 554. sy_such : 724. SymbolStr: 562, 567, 569. $sy_suppose$: 724. symPri: 1427, 1428. sy_Suppose: 1187, 1497, 1577, 1587, 1591, 1594, SymPri: 1387, 1417, 1427. 1595, 1633, 1637, 1650, 1660. SynErr: 1496, 1498, 1524, 1541, 1547, 1641, $sy_synonym$: 724. 1643, 1649. $sy_Synonym$: 1191, 1640. Synonym: 1631, 1640. sy_Take: 1191, 1587. syNotations: 720. $sy_{-}take: 724.$ syntax: 681, 735, 850, 1199, 1492. sy_That: 1181, 1191, 1237, 1239, 1564, 1567, syntax.txt: 855. 1568, 1633. syProjectivity: 720, 1233. $sy_that: 724.$ syReducibility: 720, 1275. sy_the : 724. syReflexivity: 720, 1233. sy_The: 1168, 1502, 1515, 1519, 1520, 1531, syRegistrations: 720. 1549, 1580, 1630. syRequirements: 720. sy_Then: 1173, 1218, 1497, 1580, 1587, 1589. sySch: 720, 1575. $sy_then: 724.$ sySchemes: 720.sySethood: 720, 1094, 1143, 1191, 1233, 1649. $sy_theorem$: 724. $sy_Theorem$: 1191, 1497, 1577, 1587, 1595, 1635, System T: 922. 1639, 1650, 1660. SystemTime: 122, 123.sy_Thesis: 1165, 1549, 1580. System Time To MiliSec: 121, 122, 123. $sy_thesis: 724.$ SysUtils: 25, 35, 36, 38, 43, 514. sy_Thus: 1191, 1589. sySymmetry: 720, 1233. sy_thus : 724. syThe: 720.sy_To: 1191, 1647. syTheorems: 720. $sy_-to: 724.$ syTransitivity: 720, 1233. sy_When: 1191, 1283, 1648. syT0: 724. $sy_when: 724.$ syT1: 724. $sy_where: 724.$ syT10: 724.sy_Where: 1168, 1506, 1512. syT100: 724.sy_With: 1191, 1648. syT102: 724. $sy_with: 724.$ syT103: 724. sy_wrt : 724. syT105: 724.syAbstractness: 720. syT106: 724.syAssociativity: 720, 1233. syT107: 724. syAsymmetry: 720, 1233. syT108: 724.syCoherence: 720, 1244, 1272, 1364, 1385. syT109: 724. syCommutativity: 720, 1233. *syT11*: **724**. syCompatibility: 720, 1249, 1275.syT110: 724.syT111: 724.syConnectedness: 720, 1233.syConsistency: 720, 1249. syT112: 724. syConstructors: 720. syT113: 724.

 $\S1665$ Mizar Parser INDEX 599

31000 Mizai i aisei	
syT115: 724.	syT19: 724.
syT117: 724.	syT190: 724.
syT118: 724.	syT191: 724.
syT12: 724.	syT194: 724.
syT121: 724.	syT195: 724.
syT122: 724.	syT198: 724.
syT124: 724.	syT199: 724.
syT126: 724.	syT2: 724.
syT127: 724.	syT20: 724.
syT128: 724.	syT207: 724.
syT129: 724.	syT208: 724.
syT13: 724.	syT209: 724.
syT130: 724.	syT21: 724.
syT131: 724.	syT210: 724.
syT132: 724.	syT216: 724.
syT133: 724.	syT217: 724.
syT134: 724.	syT218: 724.
syT136: 724.	syT219: 724.
syT137: 724.	syT22: 724.
syT138: 724.	syT220: 724.
syT139: 724.	syT221: 724.
syT14: 724.	syT222: 724.
syT141: 724.	syT223: 724.
syT142: 724.	syT23: 724.
syT143: 724.	syT24: 724.
syT146: 724.	$syT25: 724. \\ syT26: 724.$
syT148: 724. syT149: 724.	$syT26: 724. \\ syT27: 724.$
syT149. 724. syT15: 724.	sy127. 724. $syT28$: 724.
sy115: 724 . $sy1150$: 724 .	sy120. 724. $syT29:$ 724.
syT150: 724: $syT151: 724.$	syT3: 724.
syT152: 724.	syT30: 724.
syT153: 724 .	syT31: 724.
syT154: 724.	syT4: 724.
syT155: 724.	syT48: 724.
syT158: 724.	syT49: 724.
syT16: 724.	syT5: 724 .
syT160: 724.	syT50: 724.
syT163: 724.	syT51: 724.
syT17: 724.	syT52: 724.
syT173: 724.	syT53: 724.
syT177: 724.	syT54: 724.
syT179: 724.	syT55: 724.
syT18: 724.	syT56: 724.
syT180: 724.	syT57: 724.
syT181: 724.	syT6: 724.
syT182: 724.	syT64: 724.
syT183: 724.	syT66: 724.
syT184: 724.	syT67: 724.
syT186: 724.	syT69: 724.
syT187: 724.	syT7: 724.
syT189: 724.	syT70: 724.

syT72: 724.582, 583, 584, 596, 599, 632, 727, 1008. syT8: 724.The Art of Computer Programming: 257. syT80: 724.the (Choice operator): 790. *syT83*: **724**. $the_item: 618, 620.$ the_token: 618, 619, 620. syT84: 724.syT87: 724.Theorem: 1652, 1660. syT9: 724.TheoremListsProcessing: 61, 65, 66, 67. TheoremReference: 867, 872, 1071, 1134, 1176. syT92: 724.syT94: 724.Theorem Reference Obj: 871, 872.syT95: 724.TheoremReferencePtr: 871, 1071, 1134, 1176, syT96: 724.1376. syT98: 724.TheoremsProcessing: 61, 65, 66. syUniqueness: 720, 1385. thesis: 846. sy Vocabularies: 720.thesis_formula: $\underline{1219}$, $\underline{1458}$. $thesis_prop: 1219, 1220.$ TByteArray: 185.TCharSet: 185.ThesisFormulaObj: 846, 847. ThesisFormulaPtr: 846, 1115, 1219. TElementState: 489, 502. Temp: 319. Time: 466, 467, 471, 473. TempObject: 331. TimeMark: 85, 116, 122, 127. TIntItem: 299, 300, 301. TempString: 331.Term: 1208, 1209, 1393, 1402, 1403, 1404, 1405, TIntKeyCollection: 299, 301, 302.To_Right: 1417, 1421, 1422, 1427, 1428. 1406, 1466, 1467. Term rewriting: 1003. Token: 721, 723. Tokenise, reserved keywords: 619. Term, Bracket: 766. Term, Fraenkel: 784. Tokenization: 726. TermBegSys: 1502, 1524, 1539, 1543, 1548, TokenKind: 721, 724, 725, 726, 729, 1320, 1388, 1641, 1642. 1497, 1500, 1502, 1519, 1591, 1623, 1630. TokenName: 725, 1155, 1159, 1162, 1163, 1165, TermExpression: <u>1558</u>, 1581, 1583, 1584, 1588, 1601, 1628, 1629, 1647. 1168, 1173, 1174, 1177, 1179, 1181, 1185, 1186, 1187, 1191. TermExpressionObj: 753, 756, 758, 760, 762, 764, TokenObj: 591, 592, 601. 770, 774, 780, 782, 788, 790, 792, 794. TermKindLookupTable: 1101, 1102, 1106. TokenPtr: 591, 595, 597, 598, 599, 600, 606, 607, 612, 618, 623, 624. TermName: 1042, 1043, 1044, 1045, 1046, 1047, TokensCollection: 593, 594, 595, 596, 599, 1048, 1049, 1050, 1051, 1052, 1053, 1054, 1055, 1056, 1057, 1101. 600, 604. TermNbr: 18, 1389, 1390, 1393, 1395, 1396, 1397, TokensSet: 489, 502, 505. Total functional programming: 1399, 1402, 1403, 1404, 1405, 1406, 1409, 1413, 1429, 1430, 1443, 1445, 1456, 1459, 1460, 1461, TPath: 38.1462, 1463, 1466, 1467, 1468, 1490. TransferItems: 195, 217, 229. TermPtr: 753, 770, 771, 772, 773, 778, 779, 782, *Trim*: 517. 783, 784, 785, 786, 787, 788, 789, 816, 817, 820, trimlz: 136, 137, 143, 144, 146, 147, 148, 149, 821, 893, 894, 897, 898, 900, 901, 903, 904, 915, 150, 151. 917, 993, 994, 1003, 1004, 1014, 1045, 1068, TrimString: 27, 29, 477, 478, 558, 560, 561, 564, 1097, 1115, 1121, 1141, 1143, 1148, 1168, 1186, 565, 566, 568, 577, 1194. 1209, 1227, 1271, 1273, 1388, 1426. TrimStringLeft: 27, 28, 29.TermSort: 753, 764, 765, 770, 771, 1101, 1106. TrimStringRight: 27, 28, 29.TermSubexpression: 1506, 1512, 1515, 1516, 1517, true: 53, 65, 66, 67, 68, 69, 85, 95, 99, 108, 109, 1531, 1533, 1534, 1558. 112, 113, 139, 140, 158, 169, 263, 271, 272, TermWithArgumentsObj: 764, 765, 766, 768, 776. 337, 374, 375, 377, 380, 391, 392, 398, 421, TermWithArgumentsPtr: 764. 425, 426, 428, 530, 558, 560, 561, 583, 595, text: 107, 472, 473, 475, 476, 477, 478, 479, 485, 1096, 1138, 1143, 1168, 1193, 1195, 1197, 1207, 494, 524, 557, 562, 568, 569, 573, 576, 578, 580, 1208, 1211, 1214, 1215, 1218, 1230, 1237, 1239,

1242, 1295, 1307, 1329, 1349, 1354, 1422, 1492, uses: 5. 1500, 1586, 1594, 1596, 1597, 1622, 1623, 1633, Val: 1100, 1194. 1635, 1639, 1650, 1657, 1660. val: 139, 143, 144, 145, 146, 147, 479, 532, True: 261, 262, 270, 293, 320, 336, 354, 365, 560, 566. 421, 437, 455. ValidCharTable: 516.TRUE: 97, 98. Value: 379, 382, 385, 391, 392, 393, 397, 400, 402, TruncDir: 34, 39. 413, 429, 441, 444, 447, 460, 1422, 1428. TruncExt: 34, 40. $Try_read_ini_var$: 479. VariableIdentifier: 900, 1083, 1143, 1191, 1358. Trybulec, Andrzej: 12, 647. VariableObj: 738, 739.TSymbol: 18, 562, <u>563</u>, <u>564</u>, <u>567</u>, <u>568</u>, <u>569</u>, <u>570</u>. VariablePtr: 738, 743, 744, 893, 894, 895, 896, TSystem Time: 120, 121, 122, 123. 897, 898, 900, 901, 903, 904, 1014, 1022, 1023, Turing complete: 922. 1097, 1111, 1113, 1143, 1148, 1157, 1289, 1291, Turner, David A.: 922. 1342, 1347, 1352, 1355, 1356, 1357, 1358, 1361, TVocabulary: 573, 574, 575, 576, 578, 580. 1362, 1434, 1436, 1485, 1486. TWordArray: 185.VCXfile: 585, 586, 587. TXTStreamObj: 18, 524, <u>525</u>, <u>526</u>, <u>527</u>, 1148. VERALPHA: 80. Type, radix: 1525. VerifyPragmaIntervals: 1193, 1197, 1198. Type, Soft: 796. VerifyPragmaOff: <u>1193</u>, 1196, 1197, 1198. Type, Standard: 800. VerifyPragmaOn: <u>1193</u>, 1196, 1197, 1198. TypeChangeObj: 900, 901.VersionStr: 78, 80, 82. TypeChangePtr: 900, 1083, 1143, 1191, 1357, VER70: 187.1358. ViableFormula: <u>1549</u>, 1552, 1553, 1554, 1555, TypeChangeSort: 900, 901. 1556, 1557. TypeChangingStatementObj: 900, 901.virtual: 186, 195, 219, 235, 250, 267, 281, 286, TypeChangingStatementPtr: 900, 1083, 1143,295, 299, 307, 309, 341, 350, 359, 379, 397, 402, 1191, 1255. 418, 429, 447, 466, 469, 494, 502, 520, 524, 533, TypeExpression: 1558, 1566, 1579, 1584, 1614, 562, 571, 573, 604, 632, 693, 703, 706, 707, 710, 1616, 1621, 1624, 1649. 713, 716, 717, 743, 745, 747, 749, 751, 762, 764, TypeExpressionObj: 797, 798, 804, 806. 766, 768, 770, 772, 776, 778, 782, 784, 786, 788, TypeName: 1021, 1110. 790, 798, 800, 802, 804, 810, 812, 814, 816, 818, TypePtr: 745, 746, 786, 787, 790, 791, 797, 804, 820, 822, 824, 838, 856, 860, 864, 877, 879, 881, 805, 820, 821, 886, 887, 890, 891, 900, 901, 926, 884, 886, 888, 890, 893, 895, 897, 900, 903, 906, 927, 958, 959, 960, 961, 966, 967, 987, 988, 989, 909, 911, 913, 915, 916, 926, 930, 935, 937, 939, 990, 991, 992, 993, 994, 1001, 1002, 1014, 1021, 941, 946, 948, 950, 952, 954, 956, 958, 960, 962, 1097, 1110, 1143, 1148, 1162, 1189, 1227, 1228. 964, 966, 968, 972, 974, 976, 978, 980, 982, 984, TypeSort: 797, 798, 799. 987, 989, 991, 993, 995, 997, 999, 1001, 1003, TypeSubexpression: 1504, 1507, 1515, 1516, 1517, 1014, 1097, 1148, 1205, 1225, 1489. 1530, 1535, 1558, 1651. VocabulariesProcessing: 61, 65, 66, 67. UncapitalizeName: 1006.W: 116, 122.Unexpected: 722.WeakerThan: 379, 391, 393. UnexpectedItem: 1633, 1637, 1640, 1650, 1660. Weakly strict Mizar: 850. UnexpectedXMLElem: 487, 493. wHour: 120, 121. unit: 5. Wiedijk, Freek: 12, 796. unit_separator: <u>610</u>, 616, 638. WINBOOL: 100.Universal Formula: 1538, 1549.windows: 25, 89, 119, 463. UniversalFormulaObj: 840, 841. WIN32: 71, 77, 81, 89, 95, 96, 463. UniversalFormulaPtr: 840, 1115, 1488. Wirth, Niklaus: 5. Up: 379, 383, 429, 442.without_io_checking: 11, 46, 108, 109, 576. UpCase: 31, 70. $update_lexeme$: 498. wMilliseconds: 120, 121. $UpperCase: 30, \underline{31}, 50.$ *wMinute*: 120, 121.

word: 90, 105, 112, 113, 114, 116, 120, 124, 125, 126, 185, 187, 554, 632. Write: 580.write: 71, 72, 73, 74, 485, 529, 534, 1150, 1152, 1153. $Write_WSMizArticle: 1096.$ WriteError: 85, 86, 105, <u>107</u>, 108. WriteLn: 71, 72, 74, 76, 107, 137, 139, 140, 141, 142, 143, 144, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 473, 485, 486, 529, 534, 569, 580. writeln: 599, 1151. Wrong Word: 1498, 1515, 1520, 1526, 1527, 1531, 1548, 1549, 1556, 1572, 1574, 1576, 1580, 1598, 1604, 1605, 1609, 1612, 1628, 1657. wsAdjective: 747, 752, 1019, 1108, 1155. wsAggregateTerm: 753, 777, 1045, 1049, 1121, 1168. wsAttributiveFormula: 808, 817, 1027, 1038, 1115, 1165. wsBiconditionalFormula: 808, 833, 1027, 1032, 1116, 1163, 1165. wsBlock: 856, 859, 1207. WSBlockPtr: 858, 1014, 1017, 1145, 1148, 1187, 1202, 1205. wsBlockPtr: 856, 858, 860, 1014, 1074, 1075, 1076, 1097, 1145, 1148, 1172, 1174, 1180. wsCircumfixTerm: 753, 767, 1045, 1048, 1121, 1168. wsClusteredType: 797, 805, 1021, 1110, 1162. wsConditionalFormula: 808, 831, 1027, 1031, 1116, 1163, 1165. wsConjunctiveFormula:808, 827, 1027, 1029, 1116, 1163, 1165. wsContradiction: 808, 845, 1027, 1115, 1165. wsDisjunctiveFormula: 808, 829, 1027, 1030, 1116, 1163, 1165. wSecond: 120, 121.wsErrorFormula: 808, 849, 1027, 1105, 1115, 1165. wsErrorTerm: 753, 795, 1045, 1106, 1121, 1168. wsErrorType: 797, 807, 1021, 1162. wsExactlyTerm: 753, 789, 1045, 1121, 1168. wsExistentialFormula: 808, 843, 1027, 1041, 1115, 1165. wsFlexaryConjunctiveFormula: 808, 835, 1027, 1033, 1115, 1163, 1165. wsFlexaryDisjunctiveFormula: 808, 837, 1027, 1034, 1115, 1163, 1165. wsForgetfulFunctorTerm: 753, 779, 1045, 1051, 1121, 1168.

wsFraenkelTerm: 753, 785, 1045, 1053, 1121, 1168.

wsGlobalChoiceTerm: 753, 791, 1045, 1057, 1121, 1168. wsInfixTerm: 753, 763, 1045, 1047, 1121, 1168. wsInternalForgetfulFunctorTerm: 753, 781, 1045, 1052, 1121, 1168. wsInternalSelectorTerm: 753, 775, 1044, 1045, 1121, 1168. wsItem: 860, 861, 1207. WSItemPtr: 1014, 1079, 1080, 1095, 1146, 1148, 1191, 1202, 1205, 1225. wsItemPtr: 856, 858, 860, 1097, 1142, 1143, 1146, 1211. wsItTerm: 753, 793, 1045, 1121, 1168. wsmarticle: 850, 1199.WSMizarPrinterObj: 1148, <u>1149</u>, <u>1150</u>, <u>1151</u>, 1152, 1153, 1154, 1155, 1156, 1157, 1158, 1159, <u>1160</u>, <u>1161</u>, <u>1162</u>, <u>1163</u>, <u>1164</u>, <u>1165</u>, <u>1166</u>, <u>1167</u>, <u>1168</u>, <u>1169</u>, <u>1170</u>, <u>1171</u>, <u>1172</u>, <u>1173</u>, <u>1174</u>, <u>1175</u>, <u>1176, 1177, 1178, 1179, 1180, 1181, 1182, 1183, </u> 1184, 1185, 1186, 1187, 1188, 1189, 1190, 1191. WSMizarPrinterPtr: 1148, 1192.wsMultiPredicativeFormula: 808, 815, 1027, 1037, 1117, 1165. wsNegatedAdjective: 747, 750, 1019, 1155. wsNegatedFormula: 808, 823, 1027, 1028, 1115, 1165, 1463. wsNumeralTerm: 753, 761, 1045, 1121, 1168. wsPlaceholderTerm: 753, 759, 1045, 1046, 1121, 1168. wsPredicativeFormula: 808, 813, 1027, 1035, 1117, 1165. wsPrivateFunctorTerm: 753, 769, 1043, 1045, 1121, 1168. wsPrivatePredicateFormula: 808, 819, 1026, 1027, 1117, 1165. wsQualificationTerm: 753, 787, 1045, 1055, 1056, 1121, 1168. wsQualifyingFormula: 808, 821, 1027, 1039, 1115, 1165. wsReservedDscrType: 797. wsRightSideOfPredicativeFormula: 808, 811, 1027, 1036, 1117. wsSelectorTerm: 753, 773, 1045, 1050, 1121, 1168. wsSimpleFraenkelTerm: 753, 783, 1045, 1054, 1121, 1168. wsSimpleTerm: 753, 757, 1042, 1045, 1121, 1168. wsStandardType: 797, 801, 1021, 1110, 1162. wsStructureType: 797, 803, 1021, 1110, 1162. wsTextProper: 856, 857, 858. wsTextProperPtr: 856, 1096, 1097, 1144, 1147, 1192, 1202, 1203. WSTextProperPtr: 1014, 1016, 1148, 1188.

wsThesis: 808, 847, 1027, 1115, 1165. wsUniversalFormula: 808, 841, 1027, 1040, 1115, 1165. .wsx file: 850. x: 144.*xml_dict*: <u>462</u>, 514, 554, 588, 647, 850. *xml_inout*: <u>514</u>, 554, 588, 647, 850, 1199. *xml_match*: <u>505</u>, 506, 507, 508, 509. xml_parser: 487, 514, 647, 850. XMLASSERT: 487, 492, 676. XMLAttrKind: 462, 1101, 1104. XMLAttrName: 462, 552, 585, 586, 587, 600, 676, 677, 678, 679, 1016, 1017, 1019, 1021, 1022, 1026, 1035, 1036, 1042, 1043, 1044, 1045, 1046, 1047, 1048, 1049, 1050, 1051, 1052, 1059, 1062, 1063, 1064, 1065, 1066, 1067, 1068, 1070, 1071, 1073, 1078, 1079, 1087, 1091, 1095, 1100, 1101, 1108, 1110, 1111, 1114, 1117, 1118, 1119, 1120, 1121, 1123, 1125, 1126, 1127, 1128, 1130, 1131, 1133, 1134, 1136, 1139, 1142, $1143,\ 1144,\ 1145,\ 1146.$ XMLAttrObj: 490, 491.XMLAttrPtr: 490, 513, 530, 531, 1099, 1100. XMLElemKind: 462, 1101, 1103. XMLElemName: 462, 585, 586, 587, 600, 676, 678, 679, 1012, 1017, 1019, 1020, 1022, 1025, 1035, 1036, 1047, 1048, 1058, 1059, 1060, 1064, 1067, 1068, 1069, 1072, 1076, 1077, 1081, 1082, 1083, 1085, 1087, 1088, 1089, 1090, 1091, 1092, 1094, 1095, 1101, 1113, 1130, 1131, 1135, 1138, 1141, 1143, 1144, 1145, 1146. XMLInStreamObj: 528, 529, 530, 531, 532, 1097. XMLInStreamPtr: 528, 647, 676, 677, 1008, 1012. XMLOutStreamObj: 533, 534, 535, 536, 537, 538, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, <u>550, 551, 552, 553,</u> 600, 651, 678, 679, 1014. XMLOutStreamPtr: 533, 585. $XMLParserObj: 502, \underline{503}, \underline{504}, \underline{505}, \underline{506}, \underline{508},$ 510, 511, 512, 513, 528, 530. XMLScannObj: 494, 495, 497, 498, 501, 502. XMLTokenKind: 489, 494. XMLToStr: 514, 517, 1142, 1153. xx: 145.*XXX*: 128. *X1*: 135, 180, 294, 340, 429, 433, 437, 439, 440, 441, 442, 443, 444, 445, 461, 679. X2: 135, 180, 294, 340, 429, 433, 437, 439, 440, 441, 442, 443, 444, 445, 461, 679. y: 147.*Y1*: 294, 447, 451, 459, 460, 461. Y2: 294, 447, 451, 459, 460, 461. z: 149, 471.

ZeroPos: 103. zz: 145. z1: 133, 134, 135, 170, 171, 173, 174, 176, 177, 181. z2: 133, 134, 135, 170, 171, 173, 174, 176, 177, 181.

```
(Absolute value for an arbitrary-precision number 138) Used in section 128.
(Abstract base class for formulas 808)
                                          Used in section 737.
(Abstract base class for terms 753)
                                      Used in section 737.
(Abstract base class for types 797)
                                      Used in section 737.
(Abstract vocabulary object declaration 571)
                                                 Used in section 556.
Add token to tokens buffer and iterate 624
                                                 Used in section 622.
(Adjective (abstract syntax tree) 751)
                                          Used in section 737.
(Adjective AST constructor 752)
                                    Used in section 736.
Adjective expression (abstract syntax tree) 747
                                                     Used in section 737.
Adjective expression AST constructor 748
                                                Used in section 736.
 Aggregate term (abstract syntax tree) 776)
                                                Used in section 755.
 Arbitrary-precision rational arithmetic 160
                                                Used in section 128.
 Arithmetic for arbitrary-precision integers 143, 146, 151 \ Used in section 128.
(Assert MML version is compatible with Mizar version 481)
                                                                Used in section 479.
 Basic arithmetic operations declarations 129
                                                  Used in section 128.
 Block object implementation 694, 695, 696, 697, 698, 699, 700
 Block object interface 693 \ Used in section 688.
 BlockKinds (syntax.pas) 692
                                   Used in section 688.
Borrow 1 for Sub_{-} 145 \rangle
                           Used in section 144.
 Characters prohibited by MScanner 634 \ Used in section 633.
 Check every remaining open (left) parentheses has a corresponding partner 1522
                                                                                       Used in section 1520.
 Check for 172/173 error, goto AfterBalance if erred 1419
                                                                Used in section 1418.
 Check for 174/175 error, goto AfterBalance if erred 1420
                                                                Used in section 1418.
 Check for errors with definition items 1253
                                                Used in section 1249.
 Check identifier is not a number 622
                                          Used in section 613.
 Check if arbitrary-precision integers are zero 137
                                                       Used in section 128.
 Check if functor symbol is valid 560
                                         Used in section 558.
 Check if predicate symbol is valid 561
                                           Used in section 558.
 Check term Bl has valid functor format, goto AfterBalance if not 1425
                                                                              Used in section 1424.
 Check the popped item's linkages are valid 1276 Used in section 1249.
 Check we are redefining a mode, attribute, functor, or predicate 1636
                                                                            Used in section 1635.
 Choice term (abstract syntax tree) 790
                                             Used in section 755.
 Circumfix term (abstract syntax tree) 766
                                                Used in section 755.
 Class declaration for Item object 703
                                           Used in section 688.
 Class declarations for dicthan.pas 556 \ Used in section 554.
 Class for Within expression 1663 \ Used in section 737.
 Classes for formula (abstract syntax tree) 810, 812, 814, 816, 818, 820, 822, 824, 826, 828, 830, 832, 834, 836, 838,
    840, 842, 844, 846, 848 \ Used in section 737.
 Classes for terms (abstract syntax tree) 755
                                                 Used in section 737.
 Classes for type (abstract syntax tree) 798, 800, 802, 804, 806 \ Used in section 737.
 Close parentheses for formula 1551 \ Used in section 1549.
 Close the parentheses 1529
                                Used in section 1525.
 Communicate with items (parser.pas) 1558) Used in section 1493.
 Compare bracket symbols 663 \ Used in section 661.
 Compare formats 659
                          Used in section 650.
 Compare infix symbols 664
                               Used in sections 661 and 663.
 Compare prefix symbols 662
                                 Used in section 661.
 Compare same kinded symbols with the same number 661
                                                                 Used in section 660.
 Compare symbols of the same kind 660 \ Used in section 659.
 Complex-rational arbitrary-precision arithmetic 169
                                                          Used in section 128.
 Constants for xml_parser.pas 488 \ Used in section 487.
 Constants for common error messages reported to console 60
                                                                   Used in section 56.
```

```
(Construct the term's syntax tree after balancing arguments among subterms 1427)
                                                                                       Used in section 1417.
Constructors for derived format classess 652, 654, 656, 658
                                                              Used in section 650.
Constructors for example statements (wsmarticle.pas) 904
                                                                  Used in section 901.
 Consume "does not" or "do not", raise error otherwise 1542
                                                                   Used in section 1541.
 Consume "per cases", raise an error if they're missing 1593
                                                                  Used in section 1591.
 Count lNbr the number of dictionary entries for an article 579
                                                                    Used in section 578.
 Create a subexpression for an expression 712
                                                  Used in section 683.
 De-duplicate a string list 314
                                  Used in section 313.
 Declare XML Attribute Object 490
                                        Used in section 489.
Declare XML Parser object 502
                                    Used in section 489.
 Declare XML Scanner Object type 494
                                            Used in section 489.
(Declare FileDescrCollection data type 469)
                                               Used in section 465.
Declare FileDescr data type 466
                                     Used in section 465.
(Declare MBracketFormat object 657)
                                         Used in section 649.
Declare MFormatsList object 665
                                       Used in section 649.
Declare MFormat object 651
                                  Used in section 649.
(Declare MInfixFormat object 655)
                                      Used in section 649.
 Declare MPrefixFormat object 653
                                       Used in section 649.
⟨ Declare classes for _formats.pas 649⟩
                                         Used in section 647.
\langle Declare public comparison operators for arbitrary-precision numbers 135 \rangle Used in section 128.
(Declare public complex-valued arbitrary precision arithmetic 134)
                                                                     Used in section 128.
Declare Position as record 104
                                     Used in section 103.
\langle \text{ Delphi declaration of } CtrlSignal \text{ for Windows } 101 \rangle
                                                       Used in section 99.
(Delphi implementation of InitCtrl for Windows 98)
                                                        Used in section 96.
 Determine the ID 616
                          Used in section 613.
Emit XML for Fraenkel term (WSM) 1053
                                                Used in section 1045.
Emit XML for consider contents (WSM) 1082
                                                    Used in section 1080.
Emit XML for reconsider contents (WSM) 1083
                                                       Used in section 1080.
Emit XML for aggregate term (WSM) 1049
                                                 Used in section 1045.
Emit XML for assumptions item (WSM) 1092
                                                   Used in section 1084.
(Emit XML for attribute pattern (WSM) 1066)
                                                   Used in section 1061.
 Emit XML for attributive formula (WSM) 1038
                                                     Used in section 1027.
Emit XML for biconditional formula (WSM) 1032
                                                       Used in section 1027.
Emit XML for bracket functor pattern (WSM) 1064
                                                         Used in section 1061.
Emit XML for circumfix term (WSM) 1048
                                                Used in section 1045.
(Emit XML for conditional formula (WSM) 1031)
                                                     Used in section 1027.
 Emit XML for conjunction (WSM) 1029
                                             Used in section 1027.
\langle \text{Emit XML for definition-related items (WSM) } 1084, 1085, 1086, 1087, 1088, 1089, 1090, 1091 \rangle
                                                                                                 Used in
    section 1080.
(Emit XML for disjunction (WSM) 1030)
                                             Used in section 1027.
(Emit XML for exactly qualification term (WSM) 1056)
                                                           Used in section 1045.
Emit XML for existential formula (WSM) 1041
                                                    Used in section 1027.
(Emit XML for flexary-conjunction (WSM) 1033)
                                                     Used in section 1027.
Emit XML for flexary-disjunction (WSM) 1034
                                                    Used in section 1027.
(Emit XML for forgetful functor (WSM) 1051)
                                                 Used in section 1045.
 Emit XML for global choice term (WSM) 1057
                                                    Used in section 1045.
Emit XML for infix functor pattern (WSM) 1063
                                                      Used in section 1061.
Emit XML for infix term (WSM) 1047 Used in section 1045.
Emit XML for internal forgetful functor (WSM) 1052
                                                          Used in section 1045.
(Emit XML for mode pattern (WSM) 1065) Used in section 1061.
(Emit XML for multi-predicative formula (WSM) 1037)
                                                          Used in section 1027.
(Emit XML for negated formula (WSM) 1028) Used in section 1027.
```

```
(Emit XML for placeholder (WSM) 1046)
                                               Used in section 1045.
Emit XML for predicate pattern (WSM) 1062
                                                      Used in section 1061.
(Emit XML for predicative formula (WSM) 1035)
                                                       Used in section 1027.
 Emit XML for qualification term (WSM) 1055
                                                      Used in section 1045.
(Emit XML for qualifying formula (WSM) 1039)
                                                       Used in section 1027.
(Emit XML for registration-related items (WSM) 1093, 1094)
(Emit XML for right-side of predicative formula (WSM) 1036) Used in section 1027.
 Emit XML for schema (WSM) 1081 \ Used in section 1080.
(Emit XML for selector term (WSM) 1050)
                                                 Used in section 1045.
\langle Emit XML for simple Fraenkel term (WSM) 1054\rangle Used in section 1045.
 Emit XML for universal formula (WSM) 1040 \( \rightarrow \) Used in section 1027.
(Empty method declarations for SubexpObj 717)
                                                      Used in section 713.
 Exactly term (abstract syntax tree) 788
                                               Used in section 755.
 Example classes (wsmarticle.pas) 903 >
                                               Used in section 900.
(ExpKinds (syntax.pas) 709) Used in section 688.
(Expression class declaration 710)
                                       Used in section 688.
(Expression constructor 711) Used in section 683.
Extended block class declaration 1205
                                             Used in section 1199.
Extended block implementation 1206, 1213, 1214, 1215, 1216, 1217, 1218, 1219, 1221, 1222, 1223
                                                                                                       Used in
    section 1200.
Extended expression class declaration 1489
                                                   Used in section 1199.
(Extended expression implementation 1490, 1491)
                                                      Used in section 1200.
(Extended item class declaration 1225) Used in section 1199.
Extended item implementation 1226, 1249, 1277, 1279, 1280, 1281, 1282, 1283, 1285, 1286, 1288, 1289, 1290, 1291,
    1293, 1294, 1295, 1296, 1299, 1300, 1301, 1303, 1304, 1305, 1306, 1307, 1310, 1311, 1312, 1313, 1314, 1315, 1318, 1319,
    1321, 1322, 1323, 1324, 1325, 1326, 1327, 1329, 1330, 1332, 1333, 1334, 1335, 1337, 1339, 1341, 1342, 1343, 1344, 1345,
    1347, 1349, 1350, 1351, 1352, 1353, 1354, 1355, 1356, 1357, 1358, 1359, 1360, 1361, 1362, 1363, 1364, 1365, 1366, 1368,
    1369, 1371, 1372, 1373, 1375, 1376, 1377, 1378, 1379, 1380, 1381, 1383, 1384, 1385
(Extended subexpression class declaration 1388) Used in section 1199.
Extended subexpression implementation 1390, 1391, 1392, 1393, 1395, 1396, 1397, 1398, 1399, 1400, 1401, 1402, 1403,
    1404, 1405, 1406, 1407, 1408, 1409, 1410, 1411, 1413, 1417, 1429, 1430, 1431, 1432, 1433, 1434, 1435, 1436, 1437, 1438,
    1439, 1440, 1441, 1442, 1443, 1444, 1445, 1446, 1447, 1448, 1449, 1450, 1451, 1452, 1453, 1454, 1455, 1456, 1457, 1458,
    1459, 1460, 1461, 1462, 1463, 1464, 1465, 1466, 1467, 1468, 1469, 1470, 1471, 1472, 1473, 1474, 1475, 1476, 1477, 1478,
    1479, 1480, 1481, 1482, 1483, 1484, 1485, 1486, 1487, 1488
                                                             Used in section 1200.
\langle Find needle in MSortedList by binary search 261\rangle Used in section 260.
\langle Find needle in MSortedList by brute force 262 \rangle Used in section 260.
(Find the first significant 'sign' 635)
                                         Used in section 633.
(Flush XML output buffer, if full 539)
                                          Used in section 538.
 Forgetful functor (abstract syntax tree) 778
                                                   Used in section 755.
(Fraenkel terms (abstract syntax tree) 782, 784)
                                                     Used in section 755.
(FreePascal declaration of CtrlSignal for Windows 100) Used in section 99.
\langle FreePascal implementation of InitCtrl for Windows 97\rangle Used in section 96.
                    Used in sections 23, 54, 77, 83, 89, 102, 115, 128, 182, 462, 463, 485, 487, 514, 554, 588, 647, 681, 719,
(GNU License 4)
    735, 850, 1193, 1199, and 1492.
(Get closed subterm of Fraenkel operator or enumerated set 1512)
                                                                         Used in section 1508.
 Get closed subterm of bracketed expression 1511)
                                                       Used in section 1508.
(Get closed subterm of choice operator 1515) Used in section 1508.
(Get closed subterm of identifier 1509)
                                           Used in section 1508.
 Get closed subterm of structure 1510 >
                                            Used in section 1508.
\langle \text{ Get contents of XML start tag 507} \rangle Used in section 506.
(Get environment variable, Delphi-compatible mode 45)
                                                               Used in section 44.
(Get the identifier number for current word 1201) Used in section 1200.
```

```
(Get the next digit for dividing arbitrary-precision integers 150)
                                                                                                                                               Used in section 149.
(Get token kind based off of leading character 500)
                                                                                                                   Used in section 498.
(Global variables (_formats.pas) 648) Used in section 647.
Global variables introduced in parseraddition.pas 1202, 1210, 1212, 1227, 1231, 1234, 1240, 1243, 1247, 1256,
         1268,\ 1273,\ 1287,\ 1297,\ 1308,\ 1316,\ 1320,\ 1328,\ 1336,\ 1338,\ 1340,\ 1346,\ 1348,\ 1367,\ 1370,\ 1374,\ 1394
(Global variables publicly declared in wsmarticle.pas 1007)
                                                                                                                                          Used in section 850.
  Guess the k^{th} functor format 1424 \rangle Used in section 1418.
 Guess MizFiles from environment variables or executable path 484
                                                                                                                                                           Used in section 483.
(Handle runtime error cases for monitor.pas 91) Used in section 90.
\langle Has overflow occurred in NatFunc.Add? 395\rangle Used in section 394.
(If we matched a dictionary entry, then initialize FoundToken 620)
                                                                                                                                                        Used in section 619.
\langle \text{Implement } MFormatObj | 678 \rangle Used in section 650.
(Implementation for I/O of XML 516, 519, 521, 525, 529, 534, 540)
                                                                                                                                               Used in section 514.
(Implementation for MScanner 726, 727, 728, 729, 730, 731, 732, 733)
                                                                                                                                                  Used in section 719.
\langle \text{ Implementation for } MFormatsList 666 \rangle Used in section 650.
⟨Implementation for _formats.pas 650⟩
                                                                                             Used in section 647.
(Implementation for dicthan.pas 558, 563, 567, 572, 574, 578, 580, 581)
                                                                                                                                                         Used in section 554.
\langle Implementation for errhan.pas 106\rangle Used in section 102.
\langle \text{Implementation for librenv.pas } 467, 470, 479, 482 \rangle
                                                                                                                   Used in section 463.
(Implementation for mobjects.pas 183)
                                                                                              Used in section 182.
(Implementation for monitor.pas 90, 92)
                                                                                               Used in section 89.
(Implementation for mstate.pas 85) Used in section 83.
(Implementation for mtime.pas 118, 121)
                                                                                              Used in section 115.
(Implementation for syntax.pas 683)
                                                                                      Used in section 681.
Implementation for wsmarticle.pas 854, 857, 858, 861, 863, 865, 869, 872, 876, 878, 880, 882, 885, 887, 889, 891, 894,
         961, 963, 965, 967, 969, 971, 973, 975, 977, 979, 981, 983, 985, 988, 990, 992, 994, 996, 998, 1000, 1002, 1004, 1005, 1006,
         1008,\ 1013,\ 1015,\ 1016,\ 1017,\ 1018,\ 1019,\ 1020,\ 1021,\ 1022,\ 1023,\ 1024,\ 1025,\ 1026,\ 1027,\ 1042,\ 1043,\ 1044,\ 1045,\ 1058,\ 1044,\ 1045,\ 1044,\ 1045,\ 1044,\ 1045,\ 1044,\ 1045,\ 1044,\ 1045,\ 1044,\ 1044,\ 1045,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 1044,\ 
         1059, 1060, 1061, 1067, 1068, 1069, 1070, 1071, 1072, 1073, 1074, 1075, 1076, 1077, 1078, 1079, 1080, 1095, 1098, 1099,
         1100, 1101, 1102, 1103, 1104, 1105, 1106, 1107, 1108, 1109, 1110, 1111, 1112, 1113, 1114, 1115, 1118, 1119, 1120, 1121,
         1122, 1123, 1124, 1125, 1126, 1127, 1128, 1129, 1130, 1131, 1132, 1133, 1134, 1135, 1136, 1137, 1138, 1139, 1140, 1141,
         1142,\ 1143,\ 1144,\ 1145,\ 1146,\ 1147,\ 1149,\ 1150,\ 1151,\ 1152,\ 1153,\ 1154,\ 1155,\ 1156,\ 1157,\ 1158,\ 1159,\ 1160,\ 1161,\ 1162,
         1163, 1164, 1165, 1166, 1167, 1168, 1169, 1170, 1171, 1172, 1173, 1174, 1175, 1176, 1177, 1178, 1179, 1180, 1181, 1182,
         1183, 1184, 1185, 1186, 1187, 1188, 1189, 1190, 1191, 1192
                                                                                                                             Used in section 850.
(Implementation for mconsole.pas 65) Used in section 54.
(Implementation for scanner.pas 590, 592, 594, 595, 596, 599, 600, 602, 603, 605, 608, 609, 613, 628, 629, 631, 633, 642,
         643, 644, 645, 646 \ Used in section 588.
(Implementation of XML Parser 491, 492, 493, 495, 497, 498, 501, 503, 508, 510)
(Implementation of abstract syntax 736)
                                                                                              Used in section 735.
(Implementation of parser additions 1200) Used in section 1199.
(Implementation of parser.pas 1493, 1495, 1496, 1498, 1499, 1500, 1501) Used in section 1492.
(Implementing formula AST 811, 813, 815, 817, 819, 821, 823, 825, 827, 829, 831, 833, 835, 837, 839, 841, 843, 845, 847,
                        Used in section 736.
\langle \, \text{Implementing term AST} \,\, 757, \, 759, \, 761, \, 763, \, 765, \, 767, \, 769, \, 771, \, 773, \, 775, \, 777, \, 779, \, 781, \, 783, \, 785, \, 787, \, 789, \, 791, \, 793, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780, \, 780
                        Used in section 736.
(Implementing type AST 799, 801, 803, 805, 807)
                                                                                                           Used in section 736.
(Import units for mconsole.pas 55) Used in section 54.
(Incorrect term (abstract syntax tree) 794) Used in section 755.
(Index first character appearances among definitions 598) Used in section 596.
⟨Inference kinds (wsmarticle.pas) 874⟩
                                                                                            Used in section 867.
(Infix term (abstract syntax tree) 762) Used in section 755.
```

```
(Initialize WS Mizar article 1203)
                                        Used in section 1200.
(Initialize lKind, but exit if dictionary line contains invalid symbol 559)
                                                                                  Used in section 558.
\langle \text{ Initialize main } extBlock \text{ instance } 1208 \rangle Used in section 1206.
\langle \text{Initialize "proper text" } extBlock \text{ instance } 1211 \rangle Used in section 1206.
\langle Initialize default values for extBlock instance 1207\rangle
                                                            Used in section 1206.
(Initialize explicit priority for functor entry in dictionary 566)
                                                                       Used in section 564.
(Initialize extended item by ItemKind 1230, 1233, 1235, 1237, 1239, 1242, 1246) Used in sections 1226 and 1244.
(Initialize identifier names from .idx file 1012) Used in section 1008.
\langle Initialize symbol priorities, determine last ll, pl values 1428\rangle
                                                                      Used in section 1427.
\langle Initialize the fields for newly allocated extItem object 1229\rangle Used in section 1226.
(Initialize Bl, goto AfterBalance if term has at most one argument 1423) Used in section 1418.
(Initialize Mizfiles 483)
                            Used in section 482.
(Initialize To_Right and ArgsLength arrays 1421, 1422) Used in section 1418.
(Initilize explicit infinitive for a predicate entry in dictionary 565)
                                                                           Used in section 564.
(Int relation implementation 351)
                                        Used in section 183.
(Interface for MIZFILES library 464, 465)
                                              Used in section 463.
\langle \text{Interface for errhan.pas } 103, 105 \rangle Used in section 102.
(Interface for mstate.pas 84)
                                    Used in section 83.
(Interface for mtime.pas 116)
                                   Used in section 115.
(Interface for syntax.pas 688)
                                    Used in section 681.
(Interface for abstract syntax 737) Used in section 735.
(Interface for accommodator command line options 61)
                                                                Used in section 56.
(Interface for mconsole.pas 56)
                                    Used in section 54.
(Interface for other command line options 64) Used in section 56.
(Interface for transfer-specific command line options 63)
\langle \text{Interface for } MakeEnv \text{ command line options } 62 \rangle
                                                         Used in section 56.
(Internal forgetful functors (abstract syntax tree) 780) Used in section 755.
(Internal selector term (abstract syntax tree) 774)
                                                         Used in section 755.
(Item object implementation 704, 705, 708)
                                                Used in section 683.
\langle \text{ItemKinds (syntax.pas) } 702 \rangle Used in section 688.
(Keep parsing as long as there is an infixed operator to the right 1521)
                                                                                Used in section 1520.
(Load all tokens from the dictionary 597)
                                                Used in section 596.
(Local constants for parser.pas 1494, 1497)
                                                Used in section 1493.
(Local variables for parser additions 1209, 1228, 1232, 1238, 1241, 1244, 1245, 1248, 1252, 1258, 1260, 1262, 1269, 1271,
    1274, 1278, 1284, 1292, 1298, 1302, 1309, 1317, 1331, 1382, 1389 Used in section 1199.
(MScanner object class 632)
                                  Used in section 589.
(MToken object class 601)
                               Used in section 589.
(MTokeniser class 604)
                          Used in section 589.
(Main parse method (parser.pas) 1658) Used in section 1493.
(Mark schema proof as "skipped" 1220)
                                             Used in section 1219.
(Method declarations for Item object 707) Used in section 703.
(Methods implemented by subclasses of SubexpObj 716, 1387)
                                                                      Used in sections 717 and 1388.
(Methods overriden by extended Item class 706, 1224) Used in sections 707 and 1225.
(Modules used by mizenv.pas 25) Used in section 23.
\langle NatSeq implementation 398 \rangle Used in section 183.
(Negated adjective AST constructor 750)
                                                Used in section 736.
(Negated adjective expression (abstract syntax tree) 749)
                                                                  Used in section 737.
\langle \text{Non-windows FreePascal implementation for } InitCtrl 95 \rangle
                                                                  Used in section 94.
Numeral term (abstract syntax tree) 760 Used in section 755.
(One-argument term (abstract syntax tree) 770) Used in section 755.
Open mml.ini file 480 Used in section 479.
(Parse XML for formula with binary connective 1116) Used in section 1115.
```

Mizar Parser NAMES OF THE SECTIONS 609

```
(Parse XML for predicate-based formula 1117) Used in section 1115.
\langle \text{ Parse "equals" definiens 1628} \rangle Used in section 1625.
(Parse "means" definiens 1626) Used in section 1625.
\langle Parse "over" and any structure arguments, if any 1617\rangle Used in section 1615.
\langle \text{Parse "proof" block 1600} \rangle Used in section 1599.
\langle \text{Parse "suppose" or "case" block 1595} \rangle Used in section 1591.
(Parse "take" statement for linear reasoning 1588)
                                                        Used in section 1587.
(Parse "then" for linear reasoning 1590)
                                             Used in section 1587.
(Parse "thus" and "hence" for linear reasoning 1589) Used in section 1587.
(Parse "equals" definition-by-cases 1629) Used in section 1628.
(Parse "means" definition-by-cases 1627)
                                              Used in section 1626.
(Parse a Fraenkel operator 1513)
                                     Used in section 1512.
(Parse a "deffunc" 1581) Used in section 1580.
(Parse a "defpred" 1582)
                             Used in section 1580.
(Parse a "reconsider" statement 1584)
                                             Used in section 1580.
(Parse a "set" constant definition 1583) Used in section 1580.
(Parse an enumerated set 1514) Used in section 1512.
Parse ancestors of structure, if there are any 1616 Used in section 1615.
(Parse arguments for attribute expression 1642) Used in section 1641.
(Parse arguments to the right 1532)
                                        Used in section 1531.
(Parse bracket functor pattern 1611)
                                         Used in section 1609.
Parse comma-separated variables for quantified variables 1536
                                                                       Used in section 1535.
(Parse conditional registration cluster 1645) Used in section 1643.
(Parse contents of "suppose" block 1596)
                                              Used in section 1595.
(Parse definiens 1625) Used in section 1623.
(Parse definitions (parser.pas) 1621, 1622, 1623, 1630, 1631, 1632, 1633, 1634, 1638, 1641, 1643, 1647, 1648, 1649,
    1650, 1651, 1652, 1653 Used in section 1493.
(Parse equation or (possibly infixed) predicate 1546) Used in section 1544.
(Parse existential registration cluster 1646) Used in section 1643.
(Parse expressions (parser.pas) 1502, 1503, 1504, 1505, 1506, 1508, 1518, 1519, 1520, 1523, 1525, 1530, 1531, 1533,
    1534, 1535, 1537, 1538, 1539, 1540, 1543, 1544, 1549, 1552, 1553, 1554, 1555, 1556, 1557 Used in section 1493.
Parse field for the structure definition 1620 Used in section 1619.
Parse forgetful functor or choice of structure type 1516
                                                              Used in section 1515.
(Parse formula with "does not" or "do not" 1547) Used in section 1544.
Parse formula with "is not" or "is not" 1548 Used in section 1544.
(Parse functor registration cluster 1644) Used in section 1643.
\langle \text{ Parse infix functor pattern 1610} \rangle Used in section 1609.
(Parse item for notation block 1639) Used in section 1638.
Parse item in definition block 1635
                                         Used in section 1634.
\langle \text{ Parse iterative equations 1601} \rangle Used in section 1599.
(Parse justification for scheme 1657) Used in section 1654.
(Parse left arguments in a formula 1545) Used in section 1544.
\langle Parse library references 1573\rangle Used in section 1572.
(Parse loci, assumptions, unexpected items in a definition block 1637) Used in section 1635.
(Parse mode as radix type 1526) Used in section 1525.
(Parse mode definition 1624) Used in section 1623.
(Parse multi-predicate with "does" or "do" in copula 1541) Used in section 1540.
\langle \text{ Parse next block 1660} \rangle Used in section 1659.
(Parse optional left-paren 1528) Used in section 1525.
(Parse pattern for "set" as a mode 1605) Used in section 1604.
(Parse pattern for a mode symbols 1606) Used in section 1604.
(Parse patterns (parser.pas) 1602, 1603, 1604, 1607, 1608, 1609, 1612, 1614, 1615) Used in section 1493.
```

```
(Parse pragmas and begins 1661)
                                     Used in section 1660.
(Parse predicate pattern 1613)
                                 Used in section 1612.
                               Used in section 1549.
(Parse private formula 1550)
\langle \text{ Parse proper text } 1659 \rangle Used in section 1658.
(Parse reference to scheme from MML 1575)
                                                Used in section 1574.
(Parse scheme block (parser.pas) 1654) Used in section 1493.
(Parse scheme parameters 1655) Used in section 1654.
 Parse scheme premises 1656
                                 Used in section 1654.
Parse segment of fixed variables 1566
                                           Used in section 1565.
(Parse semicolon-separated items in a notation block 1640)
                                                                Used in section 1639.
 Parse simple Fraenkel expression or "the set" 1517 Used in section 1515.
(Parse simple justifications (parser.pas) 1571, 1574, 1576) Used in section 1493.
Parse single reference 1572
                               Used in section 1571.
(Parse start of XML tag 509)
                                Used in section 508.
(Parse statement of linear reasoning 1587) Used in section 1586.
(Parse statements and reasoning (parser.pas) 1577, 1578, 1579, 1580, 1585, 1586, 1591, 1598, 1599)
                                                                                                      Used in
    section 1493.
(Parse structure as radix type 1527) Used in section 1525.
(Parse the fields of the structure definition 1619)
                                                    Used in section 1615.
(Partial Binary integer Functions 430) Used in section 183.
(Partial integer function implementation 360) Used in section 183.
(Partial integers to Pair of integers Functions 448) Used in section 183.
(Pattern objects (wsmarticle.pas) 933, 935, 937, 939, 941) Used in section 930.
(Placeholder term (abstract syntax tree) 758) Used in section 755.
(Pop skips remaining cases 1251) Used in section 1250.
(Pop a "let" statement 1263)
                                 Used in section 1250.
(Pop a conclusion or regular statement 1261)
                                                 Used in section 1250.
(Pop a definition item 1264, 1265, 1266, 1267, 1270) Used in section 1250.
(Pop a proof step 1254, 1255, 1257, 1259)
                                           Used in section 1250.
(Pop a registration item 1272, 1275) Used in section 1250.
(Pop a token from the underlying tokens stack 630)
                                                        Used in sections 629 and 629.
(Populate global variables with XML entities 1009)
                                                        Used in section 1008.
(Populate the current line 636)
                                   Used in section 635.
(Predicate declarations for arbitrary-precision arithmetic 133)
                                                                   Used in section 128.
(Prepare to read in the contents of XML file 496) Used in section 495.
(Private functor term (abstract syntax tree) 768)
                                                      Used in section 755.
(Process "case" (local procedure) 1592)
                                           Used in section 1591.
(Process attributes (parser.pas) 1524) Used in section 1523.
(Process miscellany (parser.pas) 1559, 1560, 1561, 1562, 1563, 1564, 1565, 1567, 1568, 1569, 1570)
                                                                                                    Used in
    section 1493.
(Process post-qualified segment 1507) Used in section 1506.
(Public constants for dicthan.pas 555)
                                           Used in section 554.
(Public constants for syntax.pas 682) Used in section 688.
(Public declaration for Stream Object 520) Used in section 515.
(Public declaration for Text Stream Object 524)
                                                     Used in section 515.
(Public declaration for XML Output Stream 533)
                                                     Used in section 515.
(Public function declarations for dicthan.pas 557) Used in section 554.
(Public interface for MScanner 720, 722, 723)
                                               Used in section 719.
(Public interface for XML Input Stream 528)
                                                Used in section 515.
Public interface for mobjects.pas 184, 185, 186, 191, 194, 195, 219, 235, 250, 267, 281, 286, 295, 299, 307, 309, 340,
    341, 350, 359, 379, 397, 402, 418, 429, 447, 461 \times Used in section 182.
(Public procedures for syntax.pas 689) Used in section 688.
```

```
(Public procedures implementation for syntax.pas 684, 685, 686, 687)
                                                                          Used in section 683.
(Public variables for syntax.pas 690) Used in section 688.
Publicly declared constants in wsmarticle.pas 855
                                                           Used in section 850.
(Publicly declared functions in wsmarticle.pas 853)
                                                          Used in section 850.
Publicly declared types in wsmarticle.pas 852, 856, 862, 864, 867, 868, 870, 871, 875, 877, 879, 881, 884, 886, 888,
    890, 893, 895, 897, 900, 906, 909, 911, 913, 915, 916, 924, 926, 928, 930, 946, 948, 950, 952, 954, 956, 958, 960, 962, 964,
    966, 968, 970, 972, 974, 976, 978, 980, 982, 984, 987, 989, 991, 993, 995, 997, 999, 1001, 1003, 1014, 1097, 1148
    in section 850.
Qualified segment (abstract syntax tree) 741, 743, 745
                                                            Used in section 737.
 Qualified segment AST constructor 742, 744, 746 \ Used in section 736.
 Qualified term (abstract syntax tree) 786 \ Used in section 755.
Rational arithmetic declarations 132 \ Used in section 128.
Read formats from an XML input stream 677
(Read line into vocabulary from dictionary file 577)
                                                         Used in section 576.
 Rebalance the long term tree 1418
                                        Used in section 1417.
Replace end of long line with record separator 637
                                                         Used in section 636.
Replace every invalid character in current line with the unit character 638 \ Used in section 636.
 Report results to command line 57, 59
                                           Used in section 56.
Reset reserved keywords 1011)
                                  Used in section 1008.
(Scan for pragmas, and exit if we found one 640)
Selector term (abstract syntax tree) 772
                                              Used in section 755.
Set lResult to the index of the newly inserted string 313 \( \) Used in section 312.
Simple term (abstract syntax tree) 756
                                             Used in section 755.
(Skip all end tokens, report errors 1662)
                                             Used in section 1660.
Skip comments 641 Vsed in section 636.
(Skip whitespace for XML parser 499)
                                          Used in section 498.
(Slice pragmas 614) Used in section 613.
(Spell an error or EOF for the MTokeniser 607)
                                                     Used in section 605.
Spell an identifier for the MTokeniser 606
                                                Used in section 605.
(Stacked object implementation 308)
                                         Used in section 183.
(Store XML version of vocabulary word 1010) Used in section 1009.
String collection implementation 296
                                           Used in section 183.
String list implementation 310, 315
                                         Used in section 183.
(Subexpression constructor 714)
                                    Used in section 683.
(Subexpression destructor 715)
                                   Used in section 683.
(Subexpression object class 713)
                                    Used in section 688.
(Subexpression procedures 718)
                                    Used in section 683.
\langle Symbol for vocabulary 562\rangle Used in section 556.
Synchronize after missing 'suppose' or 'case' token 1597)
                                                                Used in section 1595.
Terms with arguments (abstract syntax tree) 764)
                                                        Used in section 755.
Test if one arbitrary-precision number is less than or equal to another 139 \ Used in section 128.
Timing utilities uses for Delphi 119 \tag{9} Used in section 118.
Timing utilities uses for FreePascal 120 \ Used in section 118.
(Token kinds for MScanner 724)
                                    Used in section 720.
 Token names for MScanner 725
                                     Used in section 722.
 Token object class 591
                            Used in section 589.
 Token type for MScanner 721
                                   Used in section 720.
 Tokens collection class 593
                               Used in section 589.
 Transform XML entity into character, if encountering an XML entity at i 518\rangle
                                                                                      Used in section 517.
Trim leading zeros from arbitrary-precision integers 136
                                                              Used in section 128.
Trim whitespace from the right of the current line 639
                                                             Used in sections 636 and 640.
(Try to find a dictionary symbol 619) Used in section 613.
```

```
Try to synchronize after failing to find initial 'case' or 'suppose' 1594
                                                                                     Used in section 1591.
(Tuples of integers 342)
⟨ Type declarations for XML I/O 515⟩
                                              Used in section 514.
Type declarations for xml_parser.pas 489)
                                                      Used in section 487.
(Type declarations for scanner 589)
                                           Used in section 588.
Types for arbitrary-precision arithmetic 130
                                                       Used in section 128.
Update content of nLastWSItem based on type of item popped 1250
                                                                                    Used in section 1249.
Update max arguments for structure symbol, if needed 1618 \ Used in section 1615.
Variable (abstract syntax tree) 738
                                             Used in section 737.
Variable AST constructor 739 \ Used in section 736.
Variables for finishing a long term in a subexpression 1426
                                                                        Used in section 1417.
Variables for slicing a phrase 612, 615, 618, 621, 623, 625
                                                                  Used in section 613.
Vocabulary object declaration 573
                                           Used in section 556.
Weakly strict Item class 860 >
                                     Used in section 856.
Whoops! ID turns out to be invalid, insert an error token, then continue 617
                                                                                             Used in section 616.
Whoops! Identifier turned out to be a number! 626 Used in section 622.
Whoops! We found an unknown token, insert a 203 error token 627
                                                                                   Used in section 613.
Windows implementation for CtrlSignal 99
                                                      Used in section 96.
(Windows implementation for InitCtrl 96)
                                                   Used in section 94.
Within expression AST implementation 1664
                                                        Used in section 736.
Write symbols to vocabulary XML file 587
                                                     Used in section 585.
Write vocabulary counts to XML file 586
                                                    Used in section 585.
(Zero and units for arbitrary-precision 131)
                                                    Used in section 128.
⟨_format.pas 647⟩
"It" term (abstract syntax tree) 792
                                              Used in section 755.
abstract_syntax.pas 735
dicthan.pas 554
\langle \text{errhan.pas } 102 \rangle
(implementation of mizenv.pas 26, 28, 31, 35, 46)
                                                         Used in section 23.
\langle \text{info.pas } 485 \rangle
\langle interface for mizenv.pas 24, 27, 30, 34\rangle Used in section 23.
(librenv.pas 463)
\langle \text{ mconsole.pas } 54 \rangle
\langle \text{ mizenv.pas } 23 \rangle
(mobjects.pas 182)
\langle \text{ monitor.pas } 89 \rangle
(mstate.pas 83)
\langle \text{ mtime.pas } 115 \rangle
(numbers.pas 128)
(parser.pas 1492)
(parseraddition.pas 1199)
(pcmizver.pas 77)
\langle \text{pragmas.pas } 1193 \rangle
\langle \text{ scanner.pas } 588, 719 \rangle
\langle \text{ syntax.pas } 681 \rangle
wsmarticle.pas 850 >
\langle \text{xml\_dict.pas } 462 \rangle
\langle \text{xml_inout.pas } 514 \rangle
\langle xml_parser.pas 487 \rangle
⟨ DisplayLine global constant 58 ⟩
                                         Used in section 57.
\langle IntSequence implementation 403 \rangle
                                          Used in section 183.
⟨ IntSet Implementation 419⟩
                                    Used in section 183.
```

613

```
\langle MCollection implementation 220 \rangle
                                                Used in section 183.
\langle MExtList \text{ implementation } 236 \rangle
                                            Used in section 183.
\langle MIntCollection implementation 300 \rangle
                                                   Used in section 183.
\langle MList \text{ implementation } 196 \rangle
                                        Used in section 183.
\langle MObject \text{ implementation } 187 \rangle
                                           Used in section 183.
⟨ MSortedCollection implementation 287 ⟩
                                                        Used in section 183.
⟨ MSortedExtList implementation 268 ⟩
                                                     Used in section 183.
⟨ MSortedList implementation 251, 263 ⟩
                                                     Used in section 183.
\langle MSortedStrList \text{ implementation } 282 \rangle
                                                    Used in section 183.
\langle MStrObj \text{ implementation } 192 \rangle
                                          Used in section 183.
\langle NatFunc \text{ implementation } 380, 396 \rangle
                                                Used in section 183.
```

Mizar Parser

	ζ	Section	Page
0	Introduction	1	1
	1. Mizar's workflow	13	4
	2. Log of todos, bugs, improvements	16	5
1	Mizar environment	23	8
2	Mizar Console	54	15
3	PC Mizar Version	77	22
4	Mizar internal state		24
5	Monitor		26
6	Error handling		30
7	Time utilities		34
8	Arbitrary precision arithmetic		37
9	Mizar Objects and Data Structures		58
	1. Base object		60
	2. Mizar String Object		62
	3. Mizar List		63
	4. Mizar Collection Class		70
	5. Simple Stacked (Extendible) Lists		73
	6. Sorted lists		77
	7. Sorted extendible lists		85
	8. Sorted list of strings		90
	9. Sorted collections		91
			93
	11. Int collections		94
	12. Stacked list of objects		96
	13. String list		98
	14. Tuples of integers		106
	15. Int rel		108
	16. Partial integers functions		111
	17. Function of natural numbers		116
	18. Sequences of natural numbers		123
	19. Integer sequences		124
	20. Integer sets		128
	21. Partial Binary integer Functions		131
	22. Partial integers to Pair of integers Functions		135
10	XML Dictionary		139
11	Environment library		140
12	Info file handling	. 485	147
13	XML Parser		149
14	I/O with XML		159
15	Vocabulary file dictionaries	. 554	168
16	Scanner		177
	1. Collections of tokens		179
	2. Mizar token objects	. 601	182
	3. Tokeniser		183
	4. Scanner Object	. 632	192
17	Format	647	106

Mizar Parser TABLE OF CONTENTS 1

	1.	List of formats	665	201
18	Syn	ıtax	681	206
	1.	Block Object	691	209
	2.	Item objects	701	211
	3.	Expressions	709	223
	4.	Subexpressions	713	224
19	MS	canner	719	232
20		stract Syntax	734	246
	1.	Terms (abstract syntax tree)	753	252
	2.	Types (abstract syntax tree)	796	263
	3.	Formulas (abstract syntax tree)	808	266
21		akly strict Mizar article	850	$\frac{275}{275}$
	1.	Weakly strict text proper	855	277
	2.	Schemes	883	286
	3.	Private definitions	892	$\frac{280}{289}$
	3. 4.	Changing types	899	$\frac{203}{291}$
	5.	Proof steps	902	$\frac{291}{293}$
	6.	Structures	$902 \\ 920$	$\frac{293}{297}$
	0. 7.	Patterns	932	301
	7. 8.			304
	-	Definitions	943	$304 \\ 317$
	9.	Registrations	986	
	10.	T T T T T T T T T T T T T T T T T T T	1005	322
	11.	8	1014	325
		O VI	1021	330
		0	1027	334
			1042	340
	10	<u> </u>	1058	347
	12.	8	1097	371
		0 71	1107	375
		5. 7. 8. 7. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.	1111	376
			1118	380
			1122	382
			1148	397
22			1193	420
23	Det		1199	423
	1.		1204	425
	2.	Extended item class		430
		1. Constructor	1226	431
		2. Popping	1249	435
		3. Registrations and notations	1277	441
		4. Processing definitions	1286	443
		5. Processing remaining statements	1335	453
	3.	Extended subexpression class	1386	466
		1. Parsing Types	1407	471
		~	1412	473
			1429	480
		0 1	1456	486
	4.		1489	495
24	Par		1492	496
=	1.		1502	500
	-	1. Terms		500
			1535	511

2	TABLE OF CONTENTS	Mizar	Parser
2.	Communication with items		519
3.	Miscellaneous		520
4.	Simple justifications	1571	524
	Statements and Reasonings		526
6.	Patterns	1602	533
	Definitions		538
	Scheme blocks		549
9.	Main parse procedure	1658	551
	erred		553
Inde	ex	1665	554