§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 "Parser module" (base/parser.pas), and all the dependencies needed to compile it. For clarity (or at least ease of reference) each "chapter" appearing in the table of contents corresponds to a different file

We are studying 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.

Paragraphs are numbered *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 but has at least one of them. Thus far, all our numbered paragraphs have consisted of "text parts" only. The "code part" can optionally 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. (This is an example of a numbered paragraph with a "named code part".)

```
\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 35, 80, 89, 127, 151, 153, 167, 185, 198, 307, 612, 617, 646, 674, 714, 748, 807, 841, 879, 895, 1030, 1373, 1379, and 1672.

2 INTRODUCTION Mizar Parser §5

5. System dependent assumptions. We will note system dependent code when appropriate, but implicitly we will assume we are working with users on a UNIX-like operating system. Hardware should not matter (barring antiques from the 1960s or earlier). We will cite the POSIX standard when appropriate, but only for the sake of discussion of things like "How long can we expect a filename to be?" (POSIX says the maximum length of a filename is Operating System dependent, but must be at least 14 bytes long.)

- **6. Asides and opinions.** Some paragraphs will be labeled as "asides" which are tangential remarks not directly relevant to understanding the code, but will enrich the reader's life. [The author will offer opinions about the design and implementation of Mizar in parenthetic sentences like this one, surrounded by double brackets.] If the reader is unsatisfied by the arbitrary opinions of a random programmer, then they can disable the asides by redefining the **\Ithink** macro to have an empty body.
- 7. 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 a module system introduced by UCSD PASCAL (c. 1977). We will need to format them for 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 (since **var**-like typography would have them appear in the index underlined).

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
```

8. 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
```

9. 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
```

10. 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
```

§11 Mizar Parser INTRODUCTION 5

11. **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
```

12. 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

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
```

13. 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
```

- 14. Logging. There appears to be a *CHReport* logger introduced in kernel/prephan.pas, but its type is defined in kernel/req_info.pas.
- 15. 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) Christoph Schwarzweller, "Mizar attributes: A technique to encode mathematical knowledge into type systems". Studies in Logic, Grammar and Rhetoric 10 no.23 (2007) 387–400.
- (5) Adam Naumowicz, "Enhanced Processing of Adjectives in Mizar". Studies in Grammar, Logic, and Rhetoric 18 no.31 (2009) 89–91 for details about the Analyzer handling attributes
- (6) 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)
- (7) Artur Korniłowicz's "On rewriting rules in Mizar" (*J. Autom. Reason.* **50** no.2 (2013) 203–210, doi:10.1007/s10817-012-9261-6)
- (8) 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 §16

Section 0.1. MIZAR'S WORKFLOW

16. 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.

- 17. 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.
- 18. 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* (§886).

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

 $\{19 \quad \text{Mizar Parser} \quad \text{MAP OF MIZAR} \quad 5$

Subsection 0.1.1. Map of Mizar

19. It will be useful to provide a summary of the files, to give the reader an idea where to find various things. We offer the following grouping of files. We will enumerate them by the chapter wherein the file is discussed.

20. System-dependent code.

- (1) base/mizenv.pas provides functions for manipulating strings and file I/O
- (2) base/pcmizver.pas contains the major and minor version for Mizar, and data about the build
- (3) base/mconsole.pas provides common functions for printing messages to the console and parsing command line optional arguments
- (4) base/errhan.pas contains the *Position* type, functions for reporting errors, writing them in particular files
- (5) base/info.pas for debugging purposes, logging to a .inf file
- (6) base/monitor.pas code for signal processing, reports errors, and when calamity strikes exit Mizar
- (7) base/mtime.pas uniform framework for timing things
- (8) base/mstate.pas code for reset the current position in an article and marking the time

21. Infrastructure for the rest of Mizar's object-oriented code.

- (9) base/numbers.pas contains code for arbitrary-precision integers, rational numbers, and rational complex numbers
- (10) base/mobjects.pas contains the common data structures used in Mizar, things like dynamic arrays and the *MObject* base class;

22. XML infrastructure.

- (11) base/xml_dict.pas contains only constant parameters and enumerated types
- (12) base/librenv.pas code for accessing the prel/subdirectories of the current article and of \$MIZFILES/
 this is only here because it defines the *MizFiles* global variable which stores the full path of the
 \$MIZFILES/ environmental variable, and *MizFiles* is needed in xml_inout.pas; [This *MizFiles* global variable should be refactored out to an earlier unit, because librenv.pas seems out of place here;]
- (13) base/xml_parser.pas provides an abstract syntax tree for XML and parses XML
- (14) base/xml_inout.pas handles reading from and writing to XML files, plus escaping strings, etc.

23. Tokenisation and other "intermediate file management".

- (15) base/dicthan.pas loading ".voc" files, and transform them into ".vct" and XML ".vcx" files
- (16) base/scanner.pas the Tokeniser and Scanner are implemented here (the naming is a little confusing, the *Scanner* class is the Tokeniser, and the *Tokeniser* class is an "abstract Tokeniser" operating on an arbitrary input stream accessed by an abstract *GetPhrase* method); also note, if we want to extend Mizar to support UTF-8 character encoding instead of ASCII, then this is the file we would modify;
- (17) base/_formats.pas contains the data structures for "formats" (basically a \(\text{Identifier}, \text{ Number of arguments to left}, \text{ Number of arguments to right} \) triple) used for parsing expressions;

24. Abstract syntax tree class hierarchies.

- (18) base/syntax.pas provides "abstract" classes Subexpression and Expression for expressions, Item and Block for statements; the actual subclasses used by the parser are in the parseraddition.pas file;
- (19) base/mscanner.pas provides a number of important global variables for the parser, ".prf" file management, as well as the *qScanner* global variable for the parser;
- (20) base/abstract_syntax.pas provides the abstract syntax tree for terms, types, attributes, formulas, and "within expressions";
- (21) base/wsmarticle.pas "weakly strict Mizar" is the name for the initial internal representation of Mizar, which has its own class hierarchy here, as well as writing a "weakly strict Mizar" article to an XML file and reading back from an XML file into a "weakly strict Mizar" abstract syntax tree;

6 MAP OF MIZAR Mizar Parser $\S25$

25. Parser "proper".

- (22) base/pragmas.pas for parsing pragmas like "::\$P-", and global variables related to them;
- (23) base/parseraddition.pas for subclasses of the syntax tree class hierarchy from syntax.pas, used for constructing a "weakly strict Mizar" AST when parsing
- (24) base/parser.pas for parsing a token stream into an abstract syntax tree

Section 0.2. LOG OF TODOS, BUGS, IMPROVEMENTS

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

27. 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 §234, GCD could be optimized to avoid calculating Mul(i,i) in every loop iteration.
- (5) In §471, 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.

- 28. 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 §433, MSortedCollection.IndexOf returns −1 when duplicate entries are allowed. Since Mizar does not seem to enable duplicate entries, this bug would never have been experienced or caught.
- (2) In §687, I think TXTStreamObj.Done needs to close the associated file.
- (3) In §723, TSymbol.Init expects an fInfinitive argument, but does not use it shouldn't it initialize $Infinite \leftarrow fInfinitive$?
- (4) In §660, escaped quotation marks are not properly handled.
- (5) For StreamObj (§680), the constructors and destructors are not virtual which would impact XMLOut-StreamObj (§693) well, we just do duplicate work in XMLOutStreamObj's constructors and destructors.
- (6) Shouldn't Tokens Collection.InitTokens (§754) invoke the inherited constructor?
- (7) Shouldn't *MTokenObj.Init* (§762) invoke inherited constructors? At least to insulate itself from changes to any of its parent (or grandparent) classes?
- (8) The constructor OutWSMizFileObj.OpenFileWithXSL (§1195) expects the XML-stylesheet located at "file://'+\$MIZFILES+'/wsmiz.xml", but that file is not present in Mizar.
- (9) In extItemObj.FinishFunctorPattern (§1503), the default case does not add a new format to the gFormatsColl dictionary.
- (10) In CreateArgs (§1573) in parseraddition.pas, when $aBase \leq 0$, this will set TermNbr to a negative number.
- (11) In the Subexpression class, there is duplicate code (§1577) 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?
- (12) In CompleteArgument (§1713), we should also test that fParenthCnt is positive, shouldn't we?
- (13) The CreateSubexpression method (§1671), for extended expression objects, may result in a memory leak when $gSubexpPtr \neq \mathbf{nil}$ that is to say, if KillSubexpression has not been invoked prior to CreateSubexpression.
- (14) 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.)
- (15) As discussed in (§1466), 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.
- (16) The gSuchThat global variable is never used anywhere (§1472)
- (17) In ATTSubexpression (§1895), 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).

- **29.** 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* (§772).
- (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?
- **30.** 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.).

31. 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.

32. 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.

10 REVIEW OF PASCAL Mizar Parser $\S 33$

Section 0.3. REVIEW OF PASCAL

33. Following Wirth's *Systematic Programming: An Introduction* (Prentice-Hall, 1973; viz., Chapter 7), we can offer the following axiomatic semantics for most of PASCAL's statements.

Assignment statements:

$$\overline{\{P[w/v]\}\ v\leftarrow w\ \{P\}}$$

Compound statements:

$${P} S_1 {Q}
 {Q} S_2 {R}
 {P} S_1; S_2 {R}$$

Conditional statements:

$$\frac{\{P \land B\} S_1 \{Q\}}{\{P \land \neg B\} S_2 \{Q\}}$$
$$\overline{\{P\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \{Q\}}$$

Simpler conditional statements:

$$\begin{array}{c} \{P \wedge B\} \ S \ \{Q\} \\ \hline \{P \wedge \neg B\} \implies \{Q\} \\ \hline \{P\} \ \text{if} \ B \ \text{then} \ S \ \{Q\} \end{array}$$

While statements:

$$\frac{\set{P \land B} S \set{P}}{\set{P} \text{ while } B \text{ do } S \set{P \land \neg B}}$$

Repeat statements:

$$\frac{ \left\{ P \right\} S \left\{ Q \right\} }{ \left\{ Q \land \neg B \right\} S \left\{ Q \right\} } \\ \overline{\left\{ P \right\} \text{ repeat } S \text{ until } B \left\{ Q \land B \right\} }$$

Selective statement (and $i = L_k$ for some k):

$$\frac{ \left\{ \begin{array}{l} P \wedge (i = L_k) \right\} S_k \left\{ Q \right\} \text{ for all } k = 1, \dots, n}{ \left\{ \begin{array}{l} P \right\} \text{ case } i \text{ of} \\ L_1 \colon S_1; \\ L_2 \colon S_2; \\ \vdots \quad \vdots \\ L_n \colon S_n; \\ \text{end}; \quad \left\{ \begin{array}{l} Q \right\} \end{array}$$

When there is no k such that $i = L_k$, the **case** statement is the same as evaluating i. We can weaken the precondition:

$$\frac{P_1 \implies P_2, \quad \{P_2\} S \{Q\}}{\{P_1\} S \{Q\}}$$

We can strengthen the postcondition (Equation (11.16), page 85, of Wirth's book):

$$\frac{Q_2 \implies Q_1, \quad \{P\} S \{Q_2\}}{\{P\} S \{Q_1\}}$$

These rules are justified as a priori valid in Chapter 5 of Wirth.

For-loops may be derived as:

34. The *exit* procedure may be invoked in a procedure or function, and it terminates the function or procedure. It roughly corresponds to C's **return** statement.

12 MIZAR ENVIRONMENT Mizar Parser §35

File 1

Mizar environment

35. We want to abstract away all the system dependent code, and provide a set of common functions Mizar will use to interact with the file system. This will include some helper functions for trimming whitespace from a String.

```
⟨ mizenv.pas 35⟩ ≡
⟨ GNU License 4⟩
unit mizenv;
interface
⟨ interface for mizenv.pas 36⟩
implementation
⟨ Modules used by mizenv.pas 37⟩
⟨ implementation of mizenv.pas 38⟩
end .
```

36. There are a few common "global variables" used by the rest of Mizar. Specifically, Mizar will be processing a file ("article"). The file may be an absolute path (e.g., /path/to/article.miz), a relative path (../article.miz), or just the filename (article.miz). In any event, we will want to refer to the filename (article.miz) as well as what Mizar calls the "article ID" (in this case, "ARTICLE" — the filename without the file extension, transformed to all capital letters).

Modern programmers may find discomfort working with global variables (and for good reason!). We remind such readers that it was common practice, until very recently, for compilers and interpreters to use global variables to describe the state of the compiler (or interpreter). We will freely refer to these global variables as "state variables", since that captures the role they play more accurately.

```
\( \) interface for mizenv.pas 36 \) \( \) \( \) \( \) \( \) \( \) MizFileName: \( string; \) \( \) the \( \) article \( \) "article \( \) miz" \\ \) \( ArticleName: \( string; \) \( \) \( \) the \( \) "article \( \) "in screaming \( snake \) \( case \) \( ArticleExt: \( string; \) \( \) \( \) "iniz" \( from \) the \( MizFileName \) \( \) \( EnvFileName: \( string; \) \( \) the \( file \) name \( given \) to \( Mizar \) as a command-line argument \( \) \( \) \( procedure \) \( See \) also \( sections 39, 41, 43, 47, 49, 51, 53, 55, 57, 59, 61, 63, \) and \( 65. \)
This code is used in section 35.
```

37. The implementation begins with various "uses". Depending on the PASCAL compiler and operating system, different libraries need to be loaded.

```
⟨ Modules used by mizenv.pas 37⟩ ≡
uses { 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 35.
```

§38 Mizar Parser MIZAR ENVIRONMENT 13

38. 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.

```
⟨ implementation of mizenv.pas 38⟩ ≡ 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 + `¬; end; See also sections 40, 42, 44, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 70, 72, 73, 74, 75, 77, and 78. This code is used in section 35.
```

39. Trimming whitespace. 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).

Remember, PASCAL is call-by-value, so the string arguments are copied when these functions are invoked. We are mutating the copy of the argument, and returning them to the user.

```
\langle \text{ interface for mizenv.pas } 36 \rangle + \equiv
function TrimStringLeft(aString: string): string;
function TrimStringRight(aString : string): string;
40. \langle \text{ implementation of mizenv.pas } 38 \rangle + \equiv
function TrimStringLeft(aString: string): string;
  begin while (length(aString) > 0) \land (aString[1] = ` ) do Delete(aString, 1, 1);
  TrimStringLeft \leftarrow aString;
  end:
function TrimStringRight(aString : string): string;
  begin while (length(aString) > 0) \land (aString[length(aString)] = `\) do
     Delete(aString, length(aString), 1);
  TrimStringRight \leftarrow aString;
  end;
     Trimming a String amounts to trimming it on the left and right.
\langle \text{ interface for mizenv.pas } 36 \rangle + \equiv
function TrimString(const aString: string): string;
42. \langle \text{ implementation of mizenv.pas } 38 \rangle + \equiv
function TrimString(const aString: string): string;
  begin TrimString \leftarrow TrimStringRight(TrimStringLeft(aString));
  end;
```

43. Uppercase and lowercase strings. We have a few more String manipulation functions for changing case, and turning an integer into a String.

```
\langle \text{interface for mizenv.pas } 36 \rangle +\equiv  function UpperCase(\mathbf{const} \ aStr: \ string): string; function MizLoCase(aChar: char): char; function LowerCase(\mathbf{const} \ aStr: \ string): string; function IntToStr(aInt: integer): string;
```

14 MIZAR ENVIRONMENT Mizar Parser $\S44$

```
44. Now, uppercase strings are obtained by uppercasing each character.
```

```
⟨ implementation of mizenv.pas 38⟩ +≡ function UpperCase (const aStr: string): string; var k: integer; { index ranging over aStr } lStr: string; { 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;
```

45. Lowercasing a String can be done by iteratively replacing each character with its lowercase version. This "lowercase a single character" function is precisely *MizLoCase*.

If the reader wished for a UTF-8 version of Mizar, then this function would require thinking very hard about how to generalize.

```
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(\mathbf{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\ \mathbf{to}\ length(aStr)\ \mathbf{do}\ lStr[i] \leftarrow MizLoCase(aStr[i]);

LowerCase \leftarrow lStr;

end;
```

46. We also want a funtion to convert an integer to a String. PASCAL provides us with a procedure.

```
function IntToStr(aInt:integer): string;
var lStr: string;
begin Str(aInt, lStr); IntToStr \leftarrow lStr;
end:
```

47. 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.

```
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{ interface for mizenv.pas } 36 \rangle + \equiv
```

```
function MFileExists(const aFileName: string): Boolean;
procedure EraseFile(const aFileName: string);
```

```
48. \langle \text{implementation of mizenv.pas } 38 \rangle + \equiv
```

```
function MFileExists(const aFileName: String): Boolean;
begin MFileExists ← FileExists(aFileName); end;
procedure EraseFile(const aFileName: String);
begin SysUtils.DeleteFile(aFileName); end;
```

 $\S49$ Mizar Parser MIZAR ENVIRONMENT 15

49. We will destructively rename a file. If a file with the name already exists, we delete it. CAUTION: This function is not used anywhere, and it appears to be buggy (the file is deleted and then renamed, which begs the question—why is it deleted?).

```
⟨interface for mizenv.pas 36⟩ +≡
procedure RenameFile(const aName1, aName2: string);

50. ⟨implementation of mizenv.pas 38⟩ +≡
procedure RenameFile(const aName1, aName2: String); {unused}
begin if MFileExists(aName1) then EraseFile(aName2);
SysUtils.RenameFile(aName1, aName2);
end;

51. Again, relying on Free Pascal's FileAge function, which returns the modification time of the file.
CAUTION: this will return a signed 32-bit integer, which will run into problems after 03:14:07 UTC on 19
January 2038 because that's 2³¹ − 1 seconds since the UNIX epoch.
⟨interface for mizenv.pas 36⟩ +≡
function GetFileTime(aFileName: string): Longint;
```

52. \langle implementation of mizenv.pas $38 \rangle + \equiv$ function GetFileTime(aFileName : String): Longint; begin $GetFileTime \leftarrow FileAge(aFileName)$; end;

 $\langle \text{ interface for mizenv.pas } 36 \rangle + \equiv$

end:

53. 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; var aDir, aName, aExt: string);

54. (implementation of mizenv.pas 38) +=
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

55. "Truncating a directory" means "throw away the directory part of the path" so we end up with just a filename and the file extension.

```
\langle interface for mizenv.pas 36 \rangle +\equiv  function TruncDir(const\ aFileName:\ string):\ string;
```

16 MIZAR ENVIRONMENT Mizar Parser §56

```
\langle \text{ implementation of mizenv.pas } 38 \rangle + \equiv
function TruncDir(const aFileName: string): string;
  var Dir, lName, Ext: string;
  begin SplitFileName(aFileName, Dir, lName, Ext); TruncDir \leftarrow lName + Ext;
  end:
      "Truncating the extension" means throwing away the extension part of a path.
\langle \text{ interface for mizenv.pas } 36 \rangle + \equiv
function TruncExt(const aFileName: string): string;
58. \langle \text{ implementation of mizenv.pas } 38 \rangle + \equiv
function TruncExt(const aFileName: string): string;
  var Dir, lName, Ext: string;
  begin SplitFileName(aFileName, Dir, lName, Ext); TruncExt \leftarrow Dir + lName;
  end:
59. Extracting the file directory will return just the directory part of a path.
\langle \text{ interface for mizenv.pas } 36 \rangle + \equiv
function ExtractFileDir(const aFileName: string): string;
60. \langle \text{ implementation of mizenv.pas } 38 \rangle + \equiv
function ExtractFileDir(const aFileName: string): string;
  var Dir, lName, Ext: string;
  begin SplitFileName(aFileName, Dir, lName, Ext); ExtractFileDir <math>\leftarrow Dir;
  end:
61. Extracting the file name will throw away both the directory and extension. For example, extracting
the file name from the path "/path/to/file.ext" gives us "file". Extracting the file extension from the
same path gives us ".ext".
\langle \text{ interface for mizenv.pas } 36 \rangle + \equiv
function ExtractFileName(const aFileName: string): string;
function ExtractFileExt(const aFileName: string): string;
62. \langle \text{ implementation of mizenv.pas } 38 \rangle + \equiv
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:
63. Changing a file name's extension. See:
  freepascal.org/docs-html/rtl/sysutils/changefileext.html
Note this does not actually change the filename in the file system, it just changes it as a string.
\langle \text{ interface for mizenv.pas } 36 \rangle + \equiv
function ChangeFileExt(const aFileName, aFileExt: string): string;
```

§64 Mizar Parser MIZAR ENVIRONMENT 17

```
\langle \text{ implementation of mizenv.pas } 38 \rangle + \equiv
function ChangeFileExt(const aFileName, aFileExt: string): string;
  begin ChangeFileExt \leftarrow SysUtils.ChangeFileExt(aFileName, aFileExt); end;
65. Environment variables. Getting an environment variable. The reader wishing to learn more about
what POSIX says about environmental variables may consult the POSIX standard, Volume 1 Chapter 8:
  pubs.opengroup.org/onlinepubs/9799919799/basedefs/V1_chap08.html
The Free PASCAL compiler handles this situation far friendlier than Delphi.
\langle \text{ interface for mizenv.pas } 36 \rangle + \equiv
function GetEnvStr(aEnvName : string): string;
66. \langle \text{ implementation of mizenv.pas } 38 \rangle + \equiv
function GetEnvStr(aEnvName : string): string;
    if_-def(FPC)
       begin GetEnvStr \leftarrow GetEnv(aEnvname); end;
    endif
    if_def (DELPHI) (Get environment variable, Delphi-compatible mode 67)
    endif
67. The Delphi-compatible version of obtaining an environment variable is rather involved: copy the string,
make it null terminated, look up the value.
\langle Get environment variable, Delphi-compatible mode 67\rangle \equiv
const cchBuffer = 255:
var lName, lpszTempPath: array [0...cchBuffer] of char;
  i: integer; lStr: string;
begin (Copy the variable name as a null-terminated string 68);
if GetEnvironmentVariable(lName, lpszTempPath, cchBuffer) > 0 then
  begin (Copy environment variable's value into lStr until we find null character 69);
  end:
GetEnvStr \leftarrow lStr;
end;
This code is used in section 66.
    \langle Copy the variable name as a null-terminated string 68 \rangle \equiv
68.
  for i \leftarrow 1 to length(aEnvname) do lName[i-1] \leftarrow aEnvname[i];
  lName[length(aEnvname)] \leftarrow #0
This code is used in section 67.
    (Copy environment variable's value into lStr until we find null character 69)
  for i \leftarrow 0 to cchBuffer do
    begin if lpszTempPath[i] = \#0 then break;
    lStr \leftarrow lStr + lpszTempPath[i];
```

This code is used in section 67.

18 MIZAR ENVIRONMENT Mizar Parser §70

70. Common printing routines. Examining a file amounts to testing if we can open the file. We close the file after opening it (because we don't want to actually want to do anything with it).

We should tweak how WEB formats a file to make it resemble the other types, rather than leave it as a "type opertor" like **array** (which is the default due to Knuth).

```
format file ≡ integer ;

⟨implementation of mizenv.pas 38⟩ +≡

procedure FileExam(const aFileName: string);

var Source: file; { the file named aFileName }

I: byte; { IOResult from trying to open the file }

begin if aFileName = ´´ then ⟨Halt: we can't open the file 71⟩;

FileMode ← 0; Assign(Source, aFileName); without_io_checking(Reset(Source)); I ← IOResult;

if I ≠ 0 then DrawIOResult(aFileName, I); { (§122) }

close(Source); FileMode ← 2;

end;

71. ⟨Halt: we can't open the file 71⟩ ≡

begin DrawMessage(´Can´´tuopenu´´u.mizu´´´, ´´); { (§118) } halt(1);

end

This code is used in section 70.
```

72. The user provides a file to Mizar as the command line argument. This typically looks like a relative path "tex/article" without any file extension. Before even trying to open "tex/article.miz", or any of the related autogenerated intermediate files, we should test the file exists.

This procedure will notify the user if the file does not exist, otherwise it is silent.

```
Again, DrawMessage comes from mconsole.pas (§118).
```

```
⟨ implementation of mizenv.pas 38 ⟩ +≡
procedure EnvFileExam(const aFileExt: string);
begin if ¬MFileExists(EnvFileName + aFileExt) then
   begin DrawMessage(`Can´´t⊔open⊔´´⊔´ + EnvFileName + aFileExt + ´⊔´´´, ´´); Halt(1);
   end;
end;
```

73. This function isn't used anywhere in Mizar. It's also misnamed: we are not "getting" the file name, we are updating the file extension if the file lacks one. A better name might be "populate missing file extension". Further, this does not test if the Nr command line argument is actually a file name or not, which is a possible source of bugs.

Remember, the *ParamCount* is PASCAL's way of counting the command-line parameters passed to the program.

```
⟨ implementation of mizenv.pas 38⟩ +≡
procedure GetFileName(ParamNr : byte; DefaultExt : string; var aFileName : string);
var lFileExt: string;
begin if ParamNr ≤ ParamCount then
   begin aFileName ← ParamStr(ParamNr); lFileExt ← ExtractFileExt(aFileName);
   if lFileExt = ``then aFileName ← ChangeFileExt(aFileName, DefaultExt);
   exit
   end;
aFileName ← ``;
end;
```

 $\S74$ Mizar Parser MIZAR ENVIRONMENT 19

74. 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. It's only used in the GetMizFileName procedure, and nowhere else in Mizar.

```
The ParamStr(Nr) returns the Nr^{th} parameter as a String (it's a PASCAL primitive).
```

```
⟨ implementation of mizenv.pas 38⟩ +≡

procedure GetFileExtName(Nr:byte; DefaultExt:string; var aFileName:string; var aFileExt:string);

begin if Nr \le ParamCount then

begin aFileName \leftarrow ParamStr(Nr); aFileExt \leftarrow ExtractFileExt(aFileName);

if aFileExt = ``then aFileExt \leftarrow DefaultExt

else aFileName \leftarrow ChangeFileExt(aFileName, ``);

exit

end;

aFileName \leftarrow ``; aFileExt \leftarrow ``;
end;
```

75. Populate the state variables using the command-line arguments. 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).

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

If there are multiple articles passed to Mizar as command-line arguments, then this function finds the first one (and uses it to populate the state variables).

A possible bug: if there are multiple files passed to Mizar and the first file passed is not a ".miz" file, then Mizar will halt as a failure instead of continuing looking for the needle in the haystack.

```
\langle \text{ implementation of mizenv.pas } 38 \rangle + \equiv
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 \text{`-'} then
        begin MizFileName \leftarrow ParamStr(i); GetFileExtName(i, aFileExt, MizFileName, ArticleExt);
        ArticleName \leftarrow ExtractFileName(MizFileName); ArticleID \leftarrow UpperCase(ArticleName);
       if \neg IsMMLIdentifier(ArticleName) then \langle Halt: invalid article name 76 \rangle;
        EnvFileName \leftarrow MizFileName; exit;
       end;
  end;
76. \langle Halt: invalid article name \frac{76}{} \rangle \equiv
  begin DrawMessage('Only_letters,_numbers_and__allowed_in_Mizar_file_names','); halt(1);
  end
This code is used in section 75.
     We will provide a standard way to populate the global variables.
\langle \text{ implementation of mizenv.pas } 38 \rangle + \equiv
```

```
⟨implementation of mizenv.pas 38⟩ +≡
procedure GetArticleName;
begin GetMizFileName('.miz');
end;
```

20 MIZAR ENVIRONMENT Mizar Parser §78

78. The second file provided to Mizar is treated as the *EnvFileName*. We need to populate the global variables if they have not been extracted from the command-line arguments already.

```
\langle \text{ implementation of mizenv.pas } 38 \rangle + \equiv
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;
79.
  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
    begin IsMMLIdentifier \leftarrow false; exit; end;
  IsMMLIdentifier \leftarrow true;
```

end;

 $\S 80$ Mizar Parser PC MIZAR VERSION 21

File 2

PC Mizar Version

80. This is used to track the version of Mizar.

```
⟨ pcmizver.pas 80⟩ ≡
⟨ GNU License 4⟩
unit pcmizver;
interface
const ⟨ Constants for pcmizver.pas 81⟩
⟨ Public functions for pcmizver.pas 84⟩
implementation
⟨ Implementation for pcmizver.pas 85⟩
end .
```

81. Note the slight variant of terminology compared to semantic versioning "Major.Minor.Patch", Mizar uses "Release.Version.Variant". This appears to be just a minor difference in vocabulary.

```
\langle \, \text{Constants for pcmizver.pas 81} \, \rangle \equiv \\ PCMizarReleaseNbr = 8; \\ PCMizarVersionNbr = 1; \\ PCMizarVariantNbr = 14; \\ \text{See also sections 82 and 83}. \\ \text{This code is used in section 80}. \\
```

82. The current year could probably be determined from the PASCAL system utilities, but it is hardcoded to 2025. The *CurrentYear* is only used in one procedure in this module, so we could easily replace it with (the possibly non-portable) *FormatDateTime*('YYYY', Now).

```
\langle \text{Constants for pcmizver.pas } 81 \rangle + \equiv Current Year = 2025;
```

83. The directory separator for the file system supports Windows and UNIX-like file systems. So Classic macOS and QNX users would have to request this changed.

Note: it might be wiser, for Free PASCAL users, to use the *DirectorySeparator* constant from the *system* unit.

```
 \begin{array}{l} \langle \ {\rm Constants} \ {\rm for} \ {\rm pcmizver.pas} \ 81 \rangle \ + \equiv \\ @\{@\&\$IFDEF \ WIN32 \ @\} \ Dir Separator = `\ `; \\ @\{@\&\$ELSE \ @\} \ Dir Separator = \ '\ '; \\ @\{@\&\$ENDIF \ @\} \end{array}
```

22 PC MIZAR VERSION Mizar Parser §84

There are only four functions provided by this module. $\langle \text{Public functions for pcmizver.pas } 84 \rangle \equiv$ **function** PCMizarVersionStr: string; function VersionStr: string; **function** PlatformNameStr: string; **function** Copyright: string; This code is used in section 80. Their implementation is relativity straightforward: just print the appropriate constants to the screen. \langle Implementation for pcmizver.pas $85 \rangle \equiv$ **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; See also sections 86, 87, and 88. This code is used in section 80. 86. $\langle \text{Implementation for pcmizver.pas } 85 \rangle + \equiv$ function VersionStr: string; var lRel, lVer, lVar: string[2]; lStr: string; **begin** Str(PCMizarReleaseNbr, lRel); Str(PCMizarVersionNbr, lVer); Str(PCMizarVariantNbr, lVar);if length(lVar) = 1 then $lVar \leftarrow 0 + lVar$; $@{@\&\$IFDEF\ VERALPHA@}lStr \leftarrow `-alpha';$ $0{0\&\$ELSE0}lStr \leftarrow :;$ @{@&\$*ENDIF*@} $VersionStr \leftarrow lRel + \text{`.'} + lVer + \text{`.'} + lVar + lStr;$ end: There are a number of platforms supported, a surprisingly large number. If we were to support more 87. 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". $\langle \text{Implementation for pcmizver.pas } 85 \rangle + \equiv$ **function** PlatformNameStr: string; var lStr: string; **begin** $lStr \leftarrow ::$ $if_def (WIN32)lStr \leftarrow lStr + `Win32`; end_if$ if_def $(LINUX)lStr \leftarrow lStr + \text{`Linux'}; end_if$ $if_def (SOLARIS) lStr \leftarrow lStr + `Solaris'; end_if$ $if_def (FREEBSD)lStr \leftarrow lStr + \text{`FreeBSD'}; end_if$ **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;

 $\S 88$ Mizar Parser PC MIZAR VERSION 23

88. The last function in the pcmizver.pas file provides a string for the Mizar version.

```
 \begin{array}{l} \langle \, {\rm Implementation \,\, for \,\, pcmizver.pas \,\, 85} \, \rangle \, + \equiv \\ {\rm function \,\,} PCMizarVersionStr \colon \, string;} \\ {\rm begin \,\,} PCMizarVersionStr \leftarrow \, {\rm `Mizar} \sqcup {\rm Ver.} \sqcup \, ' + \, VersionStr;} \\ {\rm end}; \end{array}
```

24 MIZAR CONSOLE Mizar Parser §89

File 3

Mizar Console

89. The Mizar Console unit is used for interacting with the command line. Specifically, this module will be used for printing error messages, reporting progress, and parsing command-line arguments for configuration options.

```
\langle \text{ mconsole.pas } 89 \rangle \equiv
  ⟨GNU License 4⟩
unit mconsole;
  interface (Report results to command line 108)
     (Constants for common error messages reported to console 124)
     (Interface for accommodator command line options 92)
     (Interface for MakeEnv command line options 102)
     (Interface for transfer-specific command line options 106)
     (Interface for other command line options 93)
  implementation
    ⟨Import units for mconsole.pas 90⟩
    ⟨Implementation for mconsole.pas 96⟩
  end
90. We import two modules, pcmizver and mizenv,
\langle \text{Import units for mconsole.pas } 90 \rangle \equiv
uses pcmizver, mizenv;
This code is used in section 89.
```

91. We want to have a method which allows us to flag an error (*fErrNbr*) on a given line of the article being processed. But the user may request Mizar to silence these messages. We can facilitate this by having a *DisplayLine* procedure constant.

```
\langle \textit{DisplayLine} \; \text{global constant } 91 \rangle \equiv  const \textit{DisplayLine} \colon  procedure (\textit{fLine}, \textit{fErrNbr} : integer) = NoDisplayLine; This code is used in section 108.
```

Section 3.1. PARSING COMMAND-LINE ARGUMENTS

```
92. Now, we have accommodator specific options.
\langle Interface for accommodator command line options 92 \rangle \equiv
    { Accommodator specific options: }
var SignatureProcessing, { unused }
  TheoremListsProcessing, \{unused\}
  SchemeListsProcessing, \{unused\}
  InsertHiddenFiles, { Include HIDDEN automatically? }
  FormatsProcessing: Boolean;
      { Registrations-related configuration for Accommodator }
var
  Clusters Processing, Identify Processing, Reduction Processing, Properties Processing: Boolean;
      { The environ-specifical Accommodator options }
  VocabulariesProcessing, { Accommodator will run ProcessVocabularies }
  NotationsProcessing, { Accommodator processes notations directive }
  ConstructorsProcessing, { Will the Accommodator determine which constructor to use for identifier? }
  DefinitionsProcessing, EqualitiesProcessing, ExpansionsProcessing, \{Definition environs\}
  TheoremsProcessing, SchemesProcessing: Boolean; { unused variables }
See also sections 95 and 99.
This code is used in section 89.
93. Among the state variables introduced in the mconsole unit, there is one for handling SIGINT, SIGQUIT,
and SIGTERM signals. The other UNIX signals should probably be supported, as well.
\langle Interface for other command line options 93\rangle \equiv
    { Other options: }
var CtrlCPressed: Boolean = false; { SIGINT, SIGQUIT, or SIGTERM signal received?}
  LongLines: Boolean = false; { Allow lines longer than 80 characters }
  QuietMode: Boolean = false; {Don't print anything to the console?}
  StopOnError: Boolean = false;
  FinishingPass: Boolean = false; ParserOnly: Boolean = false; \{No analyzing or checking \}
  AnalyzerOnly: Boolean = false; { Analyze, but no parsing or checking }
  CheckerOnly: Boolean = false; { Check, but do not re-analyze or re-parse }
  SwitchOffUnifier: Boolean = false;
  AxiomsAllowed: Boolean = false;
See also section 104.
This code is used in section 89.
```

- **94.** The implementation begins by initializing the Accommodator specific options. The default situation is every configuration option is *true* except for the unused variables *TheoremListsProcessing* and *SchemeListsProcessing* (both are false).
- **95.** \langle Interface for accommodator command line options $92 \rangle + \equiv$ **procedure** InitAccOptions;

Mizar Parser

 \langle Implementation for monsole.pas 96 $\rangle \equiv$ **procedure** *InitAccOptions*;

26

```
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;
end;
```

See also sections 98, 100, 103, 105, 107, 109, 111, 113, 114, 116, 119, 121, 123, and 126. This code is used in sections 89 and 117.

97. Similarly, we want to be able to reset the configuration for the accommodator make everything false. The motivation is that we want to enable only certain specific flags, and it's faster to set everything to false and then manually toggle the flags we want.

This is a private helper function for other things in the mconsole.

 \langle Implementation for mconsole.pas 96 $\rangle + \equiv$ **procedure** ResetAccOptions;

```
begin InsertHiddenFiles \leftarrow true; VocabulariesProcessinq \leftarrow false; FormatsProcessinq \leftarrow false;
NotationsProcessing \leftarrow false; \ SignatureProcessing \leftarrow false; \ ConstructorsProcessing \leftarrow false;
ClustersProcessing \leftarrow false;\ IdentifyProcessing \leftarrow false;\ ReductionProcessing \leftarrow false;
PropertiesProcessing \leftarrow false; \ DefinitionsProcessing \leftarrow false; \ EqualitiesProcessing \leftarrow false;
ExpansionsProcessing \leftarrow false; TheoremsProcessing \leftarrow false; SchemesProcessing \leftarrow false;
TheoremListsProcessing \leftarrow false; SchemeListsProcessing \leftarrow false;
end;
```

- 99. 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", "<>", "in", and "strict" will not be loaded).
- -1 will toggle LongLines to true (allowing lines with more than 80 characters)
- -q will toggle QuietMode to true
- -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. \langle Interface for accommodator command line options 92 $\rangle + \equiv$ **procedure** GetAccOptions;

```
\langle Implementation for monsole.pas 96\rangle + \equiv
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
           \forall v : \mathbf{begin} \ ResetAccOptions; \ VocabulariesProcessing \leftarrow true
           'f', 'p': begin ResetAccOptions; VocabulariesProcessing \leftarrow true; FormatsProcessing \leftarrow true;
           '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;
             end:
           h: begin InsertHiddenFiles \leftarrow false; end;
          \texttt{`l'}: LongLines \leftarrow true;
           q: QuietMode \leftarrow true;
          s: StopOnError \leftarrow true;
          endcases;
  end:
101.
        Similarly, we have MakeEnv specific options parsed from the command line flags.
        \langle \text{Interface for } MakeEnv \text{ command line options } 102 \rangle \equiv
     { MakeEnv specific options: }
\mathbf{var}\ Accomodation:\ Boolean = false;\ NewAccom:\ Boolean = false;
procedure GetMEOptions;
This code is used in section 89.
103. (Implementation for mconsole.pas 96) +\equiv
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
          'n': NewAccom \leftarrow true;
          `a`: Accomodation \leftarrow true;
          1: LongLines \leftarrow true;
           q: QuietMode \leftarrow true;
           's': StopOnError \leftarrow true;
          endcases;
  end;
```

104. The "other" options.

28

end;

```
Notably, there is a feature to allow axioms, which is completely undocumented (and probably for good reason!). The user may automatically enable axioms by placing them all in ".axm" files.
```

```
\langle Interface for other command line options 93\rangle + \equiv
procedure GetOptions;
       \langle Implementation for mconsole.pas 96\rangle + \equiv
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;
106.
       Transfer specific options.
\langle Interface for transfer-specific command line options 106\rangle \equiv
     { Transfer specific options: }
var PublicLibr: Boolean; { use "prel/\(\langle Article-name \rangle \rangle \)" subdirectory? }
procedure GetTransfOptions;
This code is used in section 89.
107. (Implementation for mconsole.pas 96) +\equiv
procedure GetTransfOptions;
  var lOption: string;
  begin PublicLibr \leftarrow false;
  if ParamCount \geq 2 then
     begin lOption \leftarrow ParamStr(2);
     if (length(lOption) = 2) \land (lOption[1] \in [',','-']) then PublicLibr \leftarrow UpCase(lOption[2]) = 'P';
```

procedure *Noise*;

Section 3.2. REPORTING RESULTS TO THE CONSOLE

```
108. We have a number of functions useful for "drawing", i.e., reporting progress and results (and so on).
\langle Report results to command line 108 \rangle \equiv
procedure InitDisplayLine(const aComment: string);
procedure NoDisplayLine(fLine, fErrNbr: integer);
⟨ DisplayLine global constant 91 ⟩
See also sections 110, 112, 115, 117, 118, 120, 122, and 125.
This code is used in section 89.
109. The qComment is used only within this module. Mizar stores the name of the pass (parser, MSM,
analyzer, checker) in gComment, which is used in a helper function to print the progress to the console.
\langle Implementation for monsole.pas 96\rangle + \equiv
var gComment: string = :: { The pass currently being run }
  disable\_io\_checking;
procedure NoDisplayLine(fLine, fErrNbr: integer);
  begin end;
procedure InitDisplayLine(const aComment: string);
  \textbf{begin} \ gComment \leftarrow aComment; \ WriteLn; \ write(aComment); \ DisplayLine \leftarrow DisplayLineInCurPos
  end:
      \langle Report results to command line 108\rangle + \equiv
procedure DrawMizarScreen(const aApplicationName: string);
procedure DrawArticleName(const fName: string);
procedure DrawStr(const aStr: string);
procedure FinishDrawing;
111. \langle Implementation for mconsole.pas 96\rangle + \equiv
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;
112. The Noise parameter rings the bell three times (the \uparrow G is Caret notation "Ctrl+G", which is ASCII
code 10 BEL). For non-Windows systems, this will write three BEL characters to the standard output stream.
Windows will do nothing.
\langle Report results to command line 108\rangle + \equiv
procedure EmptyParameterList;
```

30

end;

```
113.
       \langle Implementation for mconsole.pas 96\rangle + \equiv
procedure Noise;
  begin
  if_not_def (WIN32) write (\uparrow G \uparrow G \uparrow G); endif;
  end:
procedure EmptyParameterList;
  begin Noise; WriteLn; WriteLn('****⊔⊔Empty⊔Parameter⊔List⊔?⊔****'); halt(2);
  end;
      When the user asks Mizar to verify an article, Mizar will begin by writing to the standard output
stream "Processing: \langle Article\ name \rangle".
\langle Implementation for monsole.pas 96\rangle + \equiv
procedure DrawArticleName(const fName: string);
  begin WriteLn('Processing: __', fName); end;
      \langle Report results to command line 108 \rangle + \equiv
procedure DrawPass(const aName: string);
procedure DrawTime(const aTime: string);
procedure DrawVerifierExit(const aTime: string);
       \langle Implementation for mconsole.pas 96\rangle + \equiv
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_of_mizaring:', aTime);
  end;
       On non-Windows machines, \uparrow M is used in write to add a carriage return. Windows machines will
require #13 instead. This is because \uparrow M is "Ctrl+M" which has ASCII code 77-64=13 (see, it's the same as
#13).
\langle \text{Report results to command line } 108 \rangle + \equiv
procedure DisplayLineInCurPos(fLine, fErrNbr: integer); \langle Implementation for mconsole.pas 96 \rangle =
    procedure DisplayLineInCurPos(fLine, fErrNbr: integer);
       begin if (\neg CtrlCPressed) \land (\neg QuietMode) then
         begin write(\uparrow M + qComment + \uparrow \downarrow [\uparrow, fLine : 4);
         if fErrNbr > 0 then write(`_{\perp}*`, fErrNbr);
         write(']');
         end;
       if FinishingPass then
         begin write(` [ ], fLine : 4);
         if fErrNbr > 0 then write(`\_*`, fErrNbr);
         write(`]`);
         end;
```

118. When Mizar needs to notify the user that a critical error has occurred, *DrawMessage* will be used for communicating it. By "critical error", I mean things like Mizar cannot open the file, or there was a stack overflow, or the hard drive exploded.

```
\langle Report results to command line 108\rangle + \equiv
procedure DrawMessage(const Msg1, Msg2: string);
      \langle Implementation for mconsole.pas 96\rangle +\equiv
procedure DrawMessage(const Msg1, Msg2: string);
  var Lh: byte;
  begin Noise; WriteLn; write(****_{\bot}, Msq1); Lh \leftarrow length(Msq1);
  if length(Msg2) > Lh then Lh \leftarrow length(Msg2);
  if Lh > length(Msg1) then write(` : Lh - length(Msg1));
  WriteLn(`\_****`);
  if Msg2 \neq 1 then
    begin write("****", Msg2");
    if Lh > length(Msq2) then write( : Lh - length(Msq2));
    WriteLn(`\_****`);
    end;
  end;
      The monitor.pas file uses BuqInProcessor when reporting errors. It's a logging function for severe
situations.
\langle Report results to command line 108 \rangle + \equiv
procedure BugInProcessor;
121. \langle Implementation for mconsole.pas 96\rangle + \equiv
procedure BugInProcessor;
  begin DrawMessage('Internal_Error', ''); end;
      When reset (or rewrite) fails, Mizar will cease. We should specifically report the situation to the
user, because they can address the situation whereas we cannot.
\langle Report results to command line 108\rangle + \equiv
procedure DrawIOResult(const FileName: string; I: byte);
      \langle Implementation for mconsole.pas 96\rangle + \equiv
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_lopen_l``_l` + FileName + `_l```)
  else DrawMessage('Can' t⊔open⊔' ' + FileName + ' ', '');
  halt(1);
  end;
```

32

124. We also have a constant for error messages commonly encountered. \langle Constants for common error messages reported to console $124 \rangle \equiv$ const ErrMsg: array [1...6] of string[20] =File not found, 'Path_not_found', $\verb|`Too|| many|| open|| files|',$ 'Disk_read_error', 'Disk_write_error'); This code is used in section 89. 125. (Report results to command line 108) $+\equiv$ procedure DrawErrorsMsg(aErrorNbr : integer); 126. (Implementation for mconsole.pas 96) $+\equiv$ procedure DrawErrorsMSg(aErrorNbr : integer); begin if aErrorNbr > 0 then **begin** WriteLn; if aErrorNbr = 1 then $WriteLn(`****_{\square}1_{\square}error_{\square}detected`)$ $\mathbf{else} \ \mathit{WriteLn(\texttt{`****}_\texttt{'}}, \mathit{aErrorNbr}, \texttt{`_errors}_\mathtt{detected\texttt{'}});$ end; end;

 $\{127 \quad \text{Mizar Parser} \quad \text{ERROR HANDLING} \quad 33$

File 4

Error handling

127. There are a few common error reporting routines bundled together. We should recall Borland PASCAL's *RunError* method stops the execution of the program and generates a run-time error. The other primitive PASCAL function worth remembering is *Halt* which takes an error exit code, halts the program, and returns control to the calling program. For PASCAL specific error codes, consult:

```
https://wiki.freepascal.org/RunError

⟨errhan.pas 127⟩ ≡

⟨GNU License 4⟩

unit errhan;

interface

⟨Interface for errhan.pas 128⟩

implementation

uses mconsole, mizenv;

⟨Implementation for errhan.pas 131⟩

end;
```

128. We have a few custom types and internal variables describing the state of the Mizar error handler.

The Position type is especially important for the Parser, which will store the metadata in the abstract syntax tree. The starchy reader may wish to consult the POSIX Standard's definition (3.75 of Volume I) for "column position" which states, "Column positions are numbered starting from 1." Coincidentally, this would imply Mizar cannot properly parse files longer than $2^{31} - 1$ columns wide (or $2^{63} - 1$ columns wide for 64-bit computers).

```
⟨Interface for errhan.pas 128⟩ ≡
type Position = ⟨Declare Position as record 129⟩;
   ErrorReport = 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 sections 130, 132, 134, 139, 141, 143, 145, 147, and 149.
This code is used in section 127.

129. Position is just a pair of integers recording the line and offset ("column").
⟨Declare Position as record 129⟩ ≡
   record Line, Col: integer
   end
```

This code is used in section 128.

34 ERROR HANDLING Mizar Parser §130

130. 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.

```
The Error and ErrImm procedures are both used in the parser.
```

```
⟨Interface for errhan.pas 128⟩ +≡
procedure Error(Pos : Position; ErrNr : integer);
procedure ErrImm(ErrNr : integer);

131. ⟨Implementation for errhan.pas 131⟩ ≡
procedure Error(Pos : Position; ErrNr : integer);
begin inc(ErrorNbr);
if @PutError ≠ 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;
See also sections 133, 135, 140, 142, 144, 146, 148, and 150.
This code is used in section 127.
```

132. 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.

```
procedure WriteError(Pos : Position; ErrNr : integer);

133. ⟨Implementation for errhan.pas 131⟩ +≡
var Errors: text; { file name for errors file }
   OpenedErrors: boolean = false; { have we opened it yet? }
procedure WriteError(Pos : Position; ErrNr : integer);
   begin if ¬OpenedErrors then RunTimeError(2001);
   with Pos do WriteLn(Errors, Line, ´¬, Col, ´¬, ErrNr);
   end;
```

134. 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.
```

```
\langle Interface for errhan.pas 128\rangle +\equiv procedure OpenErrors(FileName:string);
```

 $\langle \text{Interface for errhan.pas } 128 \rangle + \equiv$

 $\{135$ Mizar Parser ERROR HANDLING 35

```
135.
       \langle \text{Implementation for errhan.pas } 131 \rangle + \equiv
procedure OpenErrors(FileName : string);
  begin if ExtractFileExt(FileName) = ``then FileName <math>\leftarrow FileName + `.err';
  Assign(Errors, FileName);
  without_io_checking(Rewrite(Errors)); { Open the FileName }
  \langle If cannot open the FileName, report an error and halt 136\rangle;
  (Initialize errhan.pas state variables 137);
  (Set current position to first line, first column 138);
  if @PutError = \mathbf{nil} \ \mathbf{then} \ PutError \leftarrow WriteError;
  end;
136. (If cannot open the FileName, report an error and halt 136) \equiv
  if IOResult \neq 0 then
    begin \ DrawMessage(`Can``t_lopen_lerrors_lfile_l``` + FileName + ```_lfor_lwriting`, ``); \ halt(1);
    end
This code is used in section 135.
137. (Initialize errhan.pas state variables 137) \equiv
  OpenedErrors \leftarrow true; ErrorNbr \leftarrow 0
This code is used in sections 135 and 140.
138. (Set current position to first line, first column 138) \equiv
  with CurPos do
    begin Line \leftarrow 1; Col \leftarrow 1
    end
This code is used in sections 135, 140, and 187.
139. Appending errors to the errors file. This isn't used anywhere in Mizar. It may be instructive for the
reader to compare this to OpenErrors.
\langle \text{Interface for errhan.pas } 128 \rangle + \equiv
procedure AppendErrors(FileName : string);
140. (Implementation for errhan.pas 131) +\equiv
procedure AppendErrors(FileName : string); { unused }
  begin if ExtractFileExt(FileName) =  then FileName \leftarrow FileName +  .err;
  Assign(Errors, FileName);
  ⟨Initialize errhan.pas state variables 137⟩;
  (Set current position to first line, first column 138);
  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
errors. This does not appear to be used anywhere in Mizar.
```

 $\langle \text{Interface for errhan.pas } 128 \rangle + \equiv$

procedure EraseErrors;

36 Error handling Mizer Perser §142

```
142.
       \langle Implementation for errhan.pas 131\rangle + \equiv
procedure EraseErrors;
  begin if OpenedErrors then
    begin OpenedErrors \leftarrow false; close(Errors); erase(Errors);
    end;
  end;
143. We can also just close the errors file. This is used in monitor.pas.
\langle \text{Interface for errhan.pas } 128 \rangle + \equiv
procedure CloseErrors:
144. \langle Implementation for errhan.pas 131 \rangle + \equiv
procedure CloseErrors;
  begin if OpenedErrors then
    begin OpenedErrors \leftarrow false; close(Errors);
  end;
145. Like I said, overflow errors are especially problematic. If/when they occur, we should just bail out
immediately. Curiously, Free PASCAL uses the 202 error for stack overflow errors, and 203 for heap overflow
errors. Mizar uses the 97 error code for overflow errors.
\langle \text{Interface for errhan.pas } 128 \rangle + \equiv
procedure OverflowError(ErrorCode: word);
146. \langle Implementation for errhan.pas 131 \rangle + \equiv
procedure OverflowError(ErrorCode : word);
  begin RTErrorCode \leftarrow ErrorCode; OverflowErroor \leftarrow true; RunError(97);
  end;
       We have an assertion utility to check if a Cond is true. When it is false, we should report a runtime
error (i.e., update RTErrorCode and invoke RunError(98)). Free PASCAL's assert function generates a 227
"Assertion failed error" error code upon failure.
\langle \text{Interface for errhan.pas } 128 \rangle + \equiv
procedure MizAssert(ErrorCode: word; Cond: Boolean);
148. \langle Implementation for errhan.pas 131 \rangle + \equiv
procedure MizAssert(ErrorCode: word; Cond: Boolean);
  begin if \neg Cond then
    begin RTErrorCode \leftarrow ErrorCode; RunError(98);
    end:
  end;
149. Last, we have a catchall for runtime errors encountered.
\langle Interface for errhan.pas 128 \rangle + \equiv
procedure RunTimeError(ErrorCode: word);
       \langle Implementation for errhan.pas 131 \rangle + \equiv
procedure RunTimeError(ErrorCode: word);
  begin RTErrorCode \leftarrow ErrorCode; RunError(99);
  end;
```

 $\S151$ Mizar Parser INFO FILE HANDLING 37

File 5

Info file handling

```
151. I don't think this is actually used anywhere, but I am including it for completeness.
\langle \text{ info.pas } 151 \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;
```

38 INFO FILE HANDLING Mizar Parser $\S152$

```
152. 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, `Mizareduarticle:u"`, MizFileName, `"`); _InfoExitProc ← ExitProc;
  ExitProc ← @InfoExitProc;
  end;
procedure CloseInfofile;
  begin close(InfoFile)
  end;
```

 $\S153$ Mizar Parser MONITOR 39

File 6

Monitor

153. There is only one single public-facing procedure in the monitor.pas file: *InitExitProc*. This just assigns the _Halt_ function (defined in this module) to the *ExitProc* pointer global variable.

```
⟨ monitor.pas 153⟩ ≡
⟨ GNU License 4⟩
unit monitor;
interface
procedure InitExitProc;
implementation
⟨ Units used by monitor.pas 154⟩;
var _ExitProc: pointer; _IOResult: integer;
⟨ Implementation for monitor.pas 155⟩
end
```

154. The monitor is used for reporting errors, which is heavily system dependent. The modules used by it are...wonky. We need the *baseunix* unit for UNIX systems, and the *windows* unit for Windows-based systems.

```
 \begin{array}{l} \langle \, \text{Units used by monitor.pas } 154 \, \rangle \equiv \\ \textbf{uses} \\ & @ \{ @ \$ IFDEF FPC @ \} \\ & @ \{ @ \$ IFNDEF WIN32 @ \} \\ & baseunix, \\ & @ \{ @ \$ ENDIF @ \} \\ & @ \{ @ \$ ENDIF @ \} \\ & mizenv, errhan, means ole \\ & @ \{ @ \$ IFDEF WIN32 @ \} \;, windows @ \{ @ \$ ENDIF @ \} \\ & \textbf{mdebug} \;\;, info \; \textbf{end\_mdebug} \\ \end{array}  This code is used in section 153.
```

40 MONITOR Mizar Parser $\S155$

```
There are a few private helper functions in this module.
\langle \text{Implementation for monitor.pas } 155 \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); (Handle runtime error cases for monitor.pas 156)
            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_loverflow_lerror`,`
        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 157.
This code is used in section 153.
```

§156 Mizar Parser MONITOR 41

```
156.
       \langle Handle runtime error cases for monitor.pas 156 \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 155.
      The MizExitProc is a private "bail out" function.
\langle Implementation for monitor.pas 155\rangle + \equiv
procedure MizExitProc:
  begin
  @\{@\&\$IFDEFIODEBUG@\}ExitProc \leftarrow \_ExitProc;
  Q{Q\&\$ELSEQ}_-Halt_-(ExitCode);
  @{@&$ENDIF@}
  end:
158. We use the MizExitProc to initialize the ExitProc pointer.
procedure InitExitProc:
  begin ExitProc \leftarrow @MizExitProc
  end;
159. Initializing Control. This is a heavily system dependent piece of code. There are two ways to
implement it (one way for Windows, another way for everyone else). Once we're done, we have to initialize
the _ExitProc and invoke InitCtrl.
  (Non-windows FreePascal implementation for InitCtrl 160)
  ⟨ Windows implementaion for InitCtrl 161⟩
  begin \_ExitProc \leftarrow ExitProc; InitCtrl;
  end.
       \langle \text{Non-windows FreePascal implementation for } InitCtrl | 160 \rangle \equiv
  Q\{Q\&\$IFDEFFPCQ\} Q\{Q\&\$IFNDEFWIN32Q\}
  procedure CatchSignal(aSig : integer); cdecl;
    begin
      case aSiq of
           SIGINT, SIGQUIT, SIGTERM: begin CtrlCPressed \leftarrow true; RunTimeError(1255); end;
      endcases;
    end;
var NewSignal, OldSigInt: SignalHandler;
procedure InitCtrl;
  begin NewSignal \leftarrow SignalHandler(@CatchSignal); OldSigInt \leftarrow fpSignal(SIGINT, NewSignal);
  OldSigInt \leftarrow fpSignal(SIGQUIT, NewSignal); OldSigInt \leftarrow fpSignal(SIGTERM, NewSignal);
  end:
    0{0\&$ENDIF}0{0\&$ENDIF}0}
This code is used in section 159.
```

42 MONITOR Mizar Parser §161

```
161. Microsoft breaks everything. This is a mess because of them.
\langle Windows implementaion for InitCtrl\ 161 \rangle \equiv
  @{@&$IFDEF WIN32@}
     (Windows implementation for CtrlSignal 164)
    @{@&$IFDEF FPC@}
       (FreePascal implementation of InitCtrl for Windows 162)
    @{@&$ENDIF @}
    @{@&$IFDEF DELPHI@}
       ⟨ Delphi implementation of InitCtrl for Windows 163⟩
    @{@&$ENDIF@}
  @{@&$ENDIF@}
This code is used in section 159.
       The FreePascal implementation is pretty succinct thanks to the libraries they provide.
\langle FreePascal implementation of InitCtrl for Windows 162\rangle \equiv
procedure InitCtrl;
  begin SetConsoleCtrlHandler(CtrlSignal, TRUE); end;
This code is used in section 161.
163. \langle Delphi implementation of InitCtrl for Windows \frac{163}{2}
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 161.
164. Windows requires a helper function CtrlSignal for this Microsoft mania.
\langle Windows implementation for CtrlSignal 164\rangle \equiv
  ⟨ FreePascal declaration of CtrlSignal for Windows 165⟩
  (Delphi declaration of CtrlSignal for Windows 166)
          { 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 161.
       \langle FreePascal declaration of CtrlSignal for Windows 165\rangle \equiv
  @{@&$IFDEF FPC@}
function CtrlSignal(aSignal: DWORD): WINBOOL; stdcall;
    @{@&$ENDIF @}
This code is used in section 164.
166. \langle Delphi declaration of CtrlSignal for Windows 166\rangle \equiv
  @{@&$IFDEF DELPHI @}
function CtrlSignal(aSignal: DWORD): BOOL; cdecl;
  @{@&$ENDIF@}
This code is used in section 164.
```

§167 Mizar Parser TIME UTILITIES 43

File 7

Time utilities

167. We will want to report to the user how much time Mizar takes during various phases of execution. This is another heavily "system dependent" library.

```
\langle \text{ mtime.pas } 167 \rangle \equiv
  \langle GNU License 4\rangle
unit mtime;
  interface
     ⟨Interface for mtime.pas 172⟩
  implementation
     ⟨Implementation for mtime.pas 168⟩
  end;
      The implementation begins with a rather thorny digression depending on which compiler we're using.
\langle \text{Implementation for mtime.pas } 168 \rangle \equiv
  ⟨Timing utilities uses for Delphi 169⟩
  ⟨Timing utilities uses for FreePascal 170⟩
See also sections 171, 173, 175, 177, 178, 180, and 184.
This code is used in section 167.
       Delphi simply requires us to introduce a constant for milliseconds.
\langle Timing utilities uses for Delphi 169\rangle \equiv
  @{@&$IFDEF DELPHI @}
```

```
uses windows;
const cmSecs = 1000;
 @{@&$ENDIF@}
```

This code is used in section 168.

44 TIME UTILITIES Mizar Parser $\S170$

170. FreePascal requires a bit more work, alas. We can use the GetTime procedure to populate the hours, minutes, seconds, and hundredths of a second. American readers forgetful of the metric system should know that 0.01 s = 10 ms (one hundredth of a second is ten milliseconds).

Note: the wMilliseconds parameter is misnamed, it does not measure in units of milliseconds but centiseconds.

```
\langle Timing utilities uses for FreePascal 170\rangle \equiv
  @{@&$IFDEF FPC@}
  uses dos;
const cmSecs = 100; \{ = 100 \text{ centiseconds per 1 second } \}
type TSystemTime =
  record wHour: word;
  wMinute: word;
  wSecond: word;
  wMilliseconds: word;
procedure GetLocalTime(var aTime : TSystemTime);
  begin with aTime do GetTime(wHour, wMinute, wSecond, wMilliseconds);
  end:
  @{@&$ENDIF@}
This code is used in section 168.
171. Now we can happily plug along implementing the functions we need. This is slightly misnamed, the
result will be centiseconds (hundredths of a second).
\langle \text{Implementation for mtime.pas } 168 \rangle + \equiv
function SystemTimeToMiliSec(const fTime: TSystemTime): longint;
  begin SystemTimeToMiliSec \leftarrow fTime.wHour * (3600 * cmSecs) +
       fTime.wMinute*longint(60*cmSecs) + fTime.wSecond*cmSecs + fTime.wMilliseconds;
  end;
       Time since we "started the clock". The result is stored in the variable W.
\langle \text{Interface for mtime.pas } 172 \rangle \equiv
procedure TimeMark(var\ W: longint);
See also sections 174, 176, 179, and 183.
This code is used in section 167.
173. \langle \text{Implementation for mtime.pas } 168 \rangle + \equiv
procedure TimeMark(\mathbf{var}\ W: longint);
  var SystemTime: TSystemTime;
  begin GetLocalTime(SystemTime); W \leftarrow SystemTimeToMiliSec(SystemTime);
  end:
```

174. When we have measured the time already W since the system started (in "milliseconds"), and we want to get the elapsed time *since we measured* W, then this function will accomplish the task. If W is greater than the lifetime of Mizar's run, then clearly something has gone awry. Mizar assumes a day has passed (in that case).

```
Note that 86400 = 24 \times 60 \times 60 is the number of minutes in one day.
```

```
\langle \, \text{Interface for mtime.pas } 172 \, \rangle \, + \equiv \\ \textbf{function } \, \textit{ElapsedTime}(W:longint) \text{: } \, \textit{longint};
```

§175 Mizar Parser TIME UTILITIES 45

```
175.
        \langle \text{Implementation for mtime.pas } 168 \rangle + \equiv
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:
176. We can transform an interval of time (in "milliseconds") into hours, minutes, seconds, a fractional
amount of time.
\langle \text{Interface for mtime.pas } 172 \rangle + \equiv
procedure MUnpackTime(W : longint; var H, M, S, F : word);
177. \langle \text{Implementation for mtime.pas } 168 \rangle + \equiv
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;
       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).
\langle Implementation for mtime.pas 168 \rangle + \equiv
function LeadingZero(w : word): String;
  var lStr: String:
  begin Str(w:0,lStr);
  if length(lStr) = 1 then lStr \leftarrow \texttt{`O'} + lStr;
  LeadingZero \leftarrow lStr;
  end:
179. Reporting time transforms a time interval (measured in milliseconds) into a human readable String.
\langle \text{Interface for mtime.pas } 172 \rangle + \equiv
function ReportTime(W : longint): String;
180. \langle \text{Implementation for mtime.pas } 168 \rangle + \equiv
function ReportTime(W : longint): String;
  var H, M, S, F: word; lTimeStr: String;
  begin MUnpackTime(ElapsedTime(W), H, M, S, F); \langle Round to nearest second 181 \rangle;
  if H \neq 0 then (Report hours and minutes 182)
  else Str(M:2, lTimeStr); { report minutes }
  ReportTime \leftarrow lTimeStr + \text{`:'} + LeadingZero(S); \{ \dots \text{ and seconds } \}
  end:
181. \langle \text{Round to nearest second } 181 \rangle \equiv
  if F \geq (cmSecs \ \mathbf{div}\ 2) then inc(S)
This code is used in section 180.
       \langle \text{ Report hours and minutes } 182 \rangle \equiv
  begin Str(H, lTimeStr); lTimeStr \leftarrow lTimeStr + `.` + LeadingZero(M)
  end
This code is used in section 180.
```

46 TIME UTILITIES Mizar Parser $\S183$

183. We also have one global variable tracking the start time of Mizar. Every time Mizar starts up, it will "mark the time" (i.e., assign to the gStartTime global variable the current time).

```
\langle Interface for mtime.pas 172 \rangle + \equiv var gStartTime: longint;
```

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

```
\langle Implementation for mtime.pas 168\rangle +\equiv begin TimeMark(gStartTime); end.
```

§185 Mizar Parser MIZAR INTERNAL STATE 47

File 8

end;

Mizar internal state

185. As far as *processing* an article, Mizar works like a "batch compiler" and works in multiple "passes". We will want to report on each "pass", informing the user how long it took or how many errors were encountered.

```
⟨ mstate.pas 185⟩ ≡
⟨ GNU License 4⟩
unit mstate;
interface
⟨ Interface for mstate.pas 186⟩
implementation
uses mizenv, pcmizver, monitor, errhan, mconsole, mtime
mdebug , info end_mdebug;
⟨ Implementation for mstate.pas 187⟩
end
```

186. We have a local (well, "module"-wide) variable *PassTime* to "start the clock" for measuring how long a pass took.

The implementation amounts to, well, these four functions. We have a couple "private" functions to help us: MError and MizarExitProc.

```
\langle Interface for mstate.pas 186 \rangle \equiv
procedure InitPass(const aPassName: string);
See also sections 188, 192, and 196.
This code is used in section 185.
187. \langle Implementation for mstate.pas _{187}\rangle \equiv
var PassTime: longint;
procedure InitPass(const aPassName: string);
  begin (Set current position to first line, first column 138);
  InitDisplayLine(aPassName); \{(\S 109)\}
  TimeMark(PassTime);
See also sections 189, 191, 193, and 197.
This code is used in section 185.
       \langle \text{Interface for mstate.pas } 186 \rangle + \equiv
procedure FinishPass;
189. \langle \text{Implementation for mstate.pas } 187 \rangle + \equiv
procedure FinishPass;
  begin FinishingPass \leftarrow true;
  if QuietMode then DisplayLine(CurPos.Line, ErrorNbr);
  FinishingPass \leftarrow false; \ DrawTime(``\lumber' + ReportTime(PassTime));
```

begin if ErrorNbr > 0 then

end: end;

begin DrawErrorsMsq(ErrorNbr); FinishDrawing;

48 **190.** We also have *MizarExitProc* as a private "helper" function. 191. \langle Implementation for mstate.pas $187 \rangle + \equiv$ **var** _ExitProc: pointer; procedure MizarExitProc; **begin** $ExitProc \leftarrow _ExitProc;$ disable_io_checking; if $IOResult \neq 0$ then; **if** ¬StopOnError **then** DisplayLine(CurPos.Line, ErrorNbr); $PutError \leftarrow WriteError; \{(\S132)\}$ DrawVerifierExit(ReportTime(gStartTime));enable_io_checking; end; 192. $\langle \text{Interface for mstate.pas } 186 \rangle + \equiv$ **procedure** InitProcessing(const aProgName, aExt: String); $\langle \text{Implementation for mstate.pas } 187 \rangle + \equiv$ **procedure** *MError*(*Pos* : *Position*; *ErrNr* : *integer*); **begin** $WriteError(Pos, ErrNr); \{(\S132)\}$ $DisplayLine(CurPos.Line, ErrorNbr); \{(\S91)\}$ end: **procedure** InitProcessing(const aProgName, aExt: string); **begin** DrawMizarScreen(aProgName); if ParamCount < 1 then EmptyParameterList; (Parse the command-line arguments for article name and options 194); $\langle \text{ Initialize the } ExitProc \ 195 \rangle;$ $PutError \leftarrow MError; OpenErrors(MizFileName);$ mdebug OpenInfoFile; end_mdebug end: 194. \langle Parse the command-line arguments for article name and options $194 \rangle \equiv$ GetArticleName; GetEnvironName; DrawArticleName(MizFileName + aExt); GetOptionsThis code is used in section 193. $\langle \text{ Initialize the } ExitProc | 195 \rangle \equiv$ $InitExitProc;\ FileExam(MizFileName + aExt);\ _ExitProc \leftarrow ExitProc;\ ExitProc \leftarrow @MizarExitProc$ This code is used in section 193. At the end, we should report the number of errors (if any were encountered). $\langle \text{Interface for mstate.pas } 186 \rangle + \equiv$ **procedure** ProcessingEnding; 197. \langle Implementation for mstate.pas $187 \rangle + \equiv$ procedure ProcessingEnding;

File 9

Arbitrary precision arithmetic

198. 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. We can easily turn this into a macro.

```
(I think we could even introduce a macro log_num(\#) \equiv debug_num(WriteLn(\#))...)
  define debug\_num(\#) \equiv @\{@\&\$IFDEFDEBUGNUM@\} \#@\{@\&\$ENDIF@\}
\langle \text{ numbers.pas } 198 \rangle \equiv
  (GNU License 4)
unit numbers;
  interface
    (Basic arithmetic operations declarations 204)
  type (Types for arbitrary-precision arithmetic 256)
  const (Zero and units for arbitrary-precision 257)
    (Rational arithmetic declarations 258)
     (Predicate declarations for arbitrary-precision arithmetic 278)
     (Declare public complex-valued arbitrary precision arithmetic 284)
    (Declare public comparison operators for arbitrary-precision numbers 302)
  implementation
  uses mizenv
    if_def (CH_REPORT), req_info, prephan, builtin endif
    mdebug , info end_mdebug;
    ⟨ Trim leading zeros from arbitrary-precision integers 200⟩
     Check if arbitrary-precision integers are zero 201
     Absolute value for an arbitrary-precision number 205
     Test if one arbitrary-precision number is less than or equal to another 206
     Arithmetic for arbitrary-precision integers 211
     (Arbitrary-precision rational arithmetic 259)
     Complex-rational arbitrary-precision arithmetic 279
  end .
```

Section 9.1. ARBITRARY-PRECISION INTEGERS

We will use "schoolbook arithmetic", representing an arbitrary precision integer as a string of digits, possibly leading with an optional sign. We will "normalize" the representation by the constraint: if the leading digit is zero, then the number is zero. So we will need to trim superfluous leading zeros.

We will also adopt the convention that the empty string is a synonym for zero.

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.

```
This will break if given a string of zeroes like a = 00, in the sense that the empty string will be returned.
\langle Trim leading zeros from arbitrary-precision integers 200\rangle \equiv
function trimlz(a : string): string;
  var i: integer;
  begin if (a = 0) \lor (a = 1) then trimlz \leftarrow a
  else begin i \leftarrow 0;
     repeat i \leftarrow i + 1;
       if a[i] \neq \text{`0'} then break;
     until i = length(a);
     trimlz \leftarrow copy(a, i, length(a));
  end;
This code is used in section 198.
201. 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 201\rangle \equiv
procedure checkzero(\mathbf{var}\ a, b: string);
  var a1, b1: string;
  begin \langle \text{Convert "}-0 \text{" into zero } 202 \rangle;
  Trim leading zeros from numerator and denominator 203;
  end:
This code is used in section 198.
202. \langle \text{Convert "}-0 \text{" into zero 202} \rangle \equiv
  if copy(a, 1, 2) = `-0` then
     begin debug_num(WriteLn(infofile, 'a=-0'));
     a \leftarrow \texttt{`0'};
```

This code is used in section 201.

if copy(b, 1, 2) = `-0' then

begin debug_num(WriteLn(infofile, 'b=-0'));

end;

 $b \leftarrow \text{`0'};$ end

```
203.
       \langle Trim leading zeros from numerator and denominator 203\rangle \equiv
  a1 \leftarrow trimlz(a);
  if a1 \neq a then
     begin debug_num(WriteLn(infofile, `ZEROS1: `, a));
     a \leftarrow a1;
     end;
  b1 \leftarrow trimlz(b);
  if b1 \neq b then
     begin debug_num(WriteLn(infofile, 'ZEROS2: ', b));
     b \leftarrow b1;
     end
This code is used in section 201.
       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). This assumes there are "double negatives"
like '--5'; the "absolute value" of '--5' would yield '-5', which should be a bug.
\langle Basic arithmetic operations declarations 204\rangle \equiv
function Abs(a:string):string;
See also sections 225, 232, 234, 236, 238, 243, 248, 250, 252, and 254.
This code is used in section 198.
205. \langle Absolute value for an arbitrary-precision number 205\rangle \equiv
function Abs(a:string):string;
  begin if length(a) > 0 then
    if a[1] = - then delete(a, 1, 1);
  Abs \leftarrow a;
  end:
This code is used in section 198.
206. 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 206\rangle \equiv
function _{-}leq(a, b: string): boolean; { compare two positive integers }
  var i, x, y, z: integer;
  begin debug_num(WriteLn(infofile, '_leq(',a,',',b,')'));
  checkzero(a,b);
  if length(a) < length(b) then \_leq \leftarrow true
  else if length(a) > length(b) then \_leq \leftarrow false
     else (Compare two positive integers with same number of digits 207);
  end;
See also sections 208, 209, and 210.
This code is used in section 198.
```

end;

```
207.
        \langle Compare two positive integers with same number of digits 207\rangle \equiv
  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;
  \_leq \leftarrow true;
  end
This code is used in section 206.
208. 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 \geq 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 leg(a,b) to determine the result.
\langle Test if one arbitrary-precision number is less than or equal to another 206\rangle + \equiv
function leg(a, b : string): Boolean;
  begin debug_num(WriteLn(infofile, `leq(`,a, `, `,b, `)`));
  checkzero(a,b);
  if a = b then leq \leftarrow true
  else begin if (a[1] = \text{`-'}) \land (b[1] \neq \text{`-'}) then leq \leftarrow true;
    if (a[1] = `-`) \land (b[1] = `-`) then leq \leftarrow \neg \_leq(abs(a), abs(b));
     if (a[1] \neq `-`) \land (b[1] = `-`) then leq \leftarrow false;
    if (a[1] \neq \text{`-'}) \land (b[1] \neq \text{`-'}) then leq \leftarrow \_leq(a,b);
     end;
  end;
      Testing if a \geq b is simply testing if b \leq a after normalizing the Strings. Mizar implements this by
(a > b) \lor (a = b), since \neg (a \le b) is identical to a > b.
\langle Test if one arbitrary-precision number is less than or equal to another 206\rangle +\equiv
function geq(a, b : string): Boolean;
  begin debug\_num(WriteLn(infofile, `geq(`,a,`,`,b,`)`));
  checkzero(a,b);
  geq \leftarrow (\neg leq(a,b)) \lor (a=b);
  end;
        Similarly, we may check if a < b by testing a \neq b and a < b.
\langle Test if one arbitrary-precision number is less than or equal to another 206\rangle +\equiv
function le(a, b : string): Boolean;
  begin debug_num(WriteLn(infofile, `le(`,a, `, `,b, `)`));
  checkzero(a,b); le \leftarrow (a \neq b) \land (leq(a,b));
  end:
function gt(a, b : string): Boolean;
  begin debug_num(WriteLn(infofile, 'gt(',a,',',b,')'));
  checkzero(a,b); gt \leftarrow \neg leq(a,b);
```

Subsection 9.1.1. Arithmetic operations

211. 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

$$\begin{array}{c}
a_m \, a_{m-1} \dots a_1 \\
+ \, b_m \, b_{m-1} \dots b_1 \\
\hline
c_{m+1} \, r_m \, r_{m-1} \dots r_1
\end{array} \tag{211.2}$$

Then we will compute

$$\frac{a_n \dots a_{m+1}}{+ c_{m+1} \atop r_{n+1} r_n \dots r_{m+1}} \tag{211.3}$$

The result is assembled from the digits $r_{n+1}r_n\cdots r_1$. \langle Arithmetic for arbitrary-precision integers 211 $\rangle \equiv$ **function** $_Add(a, b : String)$: string; var c, x, y, z, v: integer; i: integer; a1, b1, s, r: string; **begin** $\langle \text{Copy } a \text{ and } b \text{ into } a1, b1 \text{ ensuring } a1 \text{ is a longer string } 212 \rangle;$ $r \leftarrow$; $c \leftarrow 0$; **begin** \langle Add a1 and b1 as in step 1, Eq (211.2) 213 \rangle ; $\langle \text{ Carry the } c_{m+1} \text{ as in step 2, Eq (211.3) } 214 \rangle;$ end; $_Add \leftarrow trimlz(r);$ end; See also sections 215, 219, 221, 224, 226, 233, 235, 237, 239, 244, 249, 251, 253, and 255. This code is used in section 198. **212.** $\langle \text{Copy } a \text{ and } b \text{ into } a1, b1 \text{ ensuring } a1 \text{ is a longer string } 212 \rangle \equiv$ $a1 \leftarrow a; b1 \leftarrow b; debug_num(WriteLn(infofile, `_Add(`, a1, `, `, b1, `)`)); checkzero(a1, b1);$ if length(a1) < length(b1) then $\mathbf{begin}\ s \leftarrow b1\,;\ b1 \leftarrow a1\,;\ a1 \leftarrow s;$ end This code is used in section 211. **213.** $\langle \text{Add } a1 \text{ and } b1 \text{ as in step } 1, \text{ Eq } (211.2) \text{ 213} \rangle \equiv$ for $i \leftarrow 0$ to length(b1) - 1 do $\{ \text{ step } 1, \text{ Eq } (211.2) \}$ **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

This code is used in section 211.

end

This code is used in section 215.

```
\langle \text{Carry the } c_{m+1} \text{ as in step 2, Eq (211.3) 214} \rangle \equiv
  for i \leftarrow length(b1) to length(a1) - 1 do { step 2, Eq (211.3) }
     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
This code is used in section 211.
        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.
This means the result is always non-negative.
\langle Arithmetic for arbitrary-precision integers 211\rangle + \equiv
function \_Sub(a, b : string): string;
  \mathbf{var}\ x, y, z, v:\ integer;\ i:\ integer;\ a1, b1, s, r:\ string;
     \langle "Borrow 1" procedure for \_Sub \ 216 \rangle
  begin a1 \leftarrow a; b1 \leftarrow b;
  debug_num(WriteLn(infofile, `_Sub(`, a1, `, `, b1, `)`));
  checkzero (a1, b1); (Swap a1 and b1 if b1 \le a1 217);
  r \leftarrow ``;
     begin
        for i \leftarrow 0 to length(b1) - 1 do (Subtract the i^{th} digit of b1 from a1 218);
        for i \leftarrow length(a1) - length(b1) downto 1 do { nothing left to subtract }
           begin r \leftarrow a1[i] + r; end; { so copy remaining digits of minuend }
     end;
  \_Sub \leftarrow trimlz(r);
end;
       This is a private "helper function" for subtraction.
\langle "Borrow 1" procedure for \_Sub \ 216 \rangle \equiv
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;
This code is used in section 215.
217. \langle \text{Swap } a1 \text{ and } b1 \text{ if } b1 \langle a1 \text{ 217} \rangle \equiv
  if \neg leq(b1, a1) then
     begin s \leftarrow b1; b1 \leftarrow a1; a1 \leftarrow s;
```

218. We compute v = x - y where $x \leftarrow (a1)_i$ (possibly borrowing from the next digit of a1) and $y \leftarrow (b1)_i$. We store this as the next digit in the result r.

```
 \begin{split} &\langle \operatorname{Subtract\ the}\ i^{\operatorname{th}}\ \operatorname{digit\ of}\ b1\ \operatorname{from}\ a1\ \ 218 \rangle \equiv \\ & \mathbf{begin}\ val(a1[\operatorname{length}(a1)-i],x,z);\ val(b1[\operatorname{length}(b1)-i],y,z); \\ & \mathbf{if}\ x < y\ \mathbf{then} \\ & \mathbf{begin}\ borrow(\operatorname{length}(a1)-i);\ x \leftarrow x+10;\ \mathbf{end}; \\ & v \leftarrow x-y;\ Str(v,s);\ r \leftarrow s+r; \\ & \mathbf{end} \end{split}
```

This code is used in section 215.

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

```
\langle Arithmetic for arbitrary-precision integers 211\rangle + \equiv
function \_Mul1(a:string; y:integer): string;
   \mathbf{var}\ c, x, z, v:\ integer;\ i:\ integer;\ s, r:\ string;
   begin debug_num(WriteLn(infofile, `_Mul1(`, a, `, `, y, `)`));
   r \leftarrow  ; c \leftarrow 0;
   for i \leftarrow 0 to length(a) - 1 do \langle Multiply i^{th} \text{ digit of } a \text{ by } y \text{ 220} \rangle;
   if c \neq 0 then
      begin Str(c,s); r \leftarrow s + r;
      end;
   \_mul1 \leftarrow trimlz(r);
   end:
220. \langle \text{Multiply } i^{\text{th}} \text{ digit of } a \text{ by } y \text{ 220} \rangle \equiv
   begin val(a[length(a) - i], x, z);
   if x * y + c > 9 then
      begin v \leftarrow (x * y + c) \bmod 10; c \leftarrow (x * y + c) \operatorname{div} 10;
   else begin v \leftarrow x * y + c; c \leftarrow 0;
      end;
   Str(v,s); r \leftarrow s + r;
   end
```

This code is used in section 219.

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

```
\langle Arithmetic for arbitrary-precision integers 211\rangle +\equiv function \_Mul(a,b:string): string; var\ y,z: integer; i,j: integer; a1,b1,s,r: string; begin\ \langle Copy a into a1 and b into b1, ensuring b1 is a shorter string 222\rangle; r\leftarrow ^{\circ}0^{\circ}; for i\leftarrow0 to length(b1)-1 do \langle Multiply i^{th} digit of b1 to a1 and add it to r 223\rangle; \_Mul\leftarrow trimlz(r); end;
```

```
(Get the next digit for dividing arbitrary-precision integers 229)
   begin debug_num(WriteLn(infofile, `_Div(`, a, `, `, b, `)`));
   checkzero(a,b);
  if a = b then _{-}div \leftarrow 11
   else if \neg leg(b, a) then _{\mathbf{div}} \leftarrow \mathbf{0}
   else \langle \text{Long division of } a \text{ by } b \text{ 227} \rangle;
end;
```

227. We take the leading digits of a and treat them as a new integer $s = a_1 \cdots a_z$. We only take as many digits necessary to make $b \leq s$ but with $a_1 \cdots a_{z-1} < b$. Then we compute rs such that $s = b \times rs + r$ for some $0 \leq r < b$. We update $s \leftarrow s - rs \times b$ and move to the next digit of a (updating s) using the gets function. This reflects "long division" as taught in gradeschool.

```
\langle Long division of a by b 227\rangle \equiv
   begin s \leftarrow \ \ \ ; \ r \leftarrow \ \ \ ; \ z \leftarrow 1;
   for i \leftarrow 1 to length(b) do s \leftarrow s + a[i]; { copy leading digits of a into s}
   \langle \text{Ensure } b \leq s \text{ by adding another digit of } a, \text{ initialize } z \text{ 228} \rangle; \quad \{z \leftarrow length(s)\}
   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
This code is used in section 226.
228. Ensure b \leq s by adding another digit of a, initialize z \geq 228
   if leq(b,s) then z \leftarrow length(b)
   else begin s \leftarrow s + a[length(b) + 1]; z \leftarrow length(b) + 1; end
This code is used in section 227.
229. We just need to "get the next digit" of a, if available, and append it to s.
   define remaining\_digits\_are\_zero \equiv (trimlz(copy(a, z + c, length(a))) = `0`)
\langle Get the next digit for dividing arbitrary-precision integers 229 \rangle \equiv
procedure gets;
   var j: integer;
   begin c \leftarrow 1; s \leftarrow Sub(s, mul(rs, b)); {i.e., s \leftarrow s \bmod b}
   if (s = 0) \land remaining\_digits\_are\_zero then
      \langle \text{Copy remainder of } a \text{ into } s, \text{ and terminate the function } 230 \rangle;
  if z + 1 \le length(a) then \langle Append next digit of a onto s, incrementing c 231\rangle;
   while (\neg leq(b,s)) \land (z+c \leq length(a)) do \langle Append next digit of a onto s, incrementing c = 231 \rangle;
   z \leftarrow z + c - 1;
  end; \{gets\}
This code is used in section 226.
230. (Copy remainder of a into s, and terminate the function 230) \equiv
   begin debug\_num(WriteLn(infofile, `Rewriting_zeros: `, <math>copy(a, z + c, length(a))));
  r \leftarrow r + copy(a, z + c, length(a)); exit;
   end
This code is used in section 229.
231. \langle Append next digit of a onto s, incrementing c 231\rangle \equiv
   begin s \leftarrow s + a[z+c]; inc(c);
   if (\neg leq(b, s)) then r \leftarrow r + `0`; {shortcut: division will add a zero digit anyways}
   end
This code is used in sections 229 and 229.
```

232. Modulo. We can compute $a \mod b$ by observing if a < b then we should obtain a. Otherwise, we should compute $q \leftarrow a \operatorname{div} b$, then a - qb is $a \mod b$.

```
\langle Basic arithmetic operations declarations 204\rangle +\equiv function \_Mod(a, b : string): string;
```

```
\langle Arithmetic for arbitrary-precision integers 211\rangle + \equiv
function \_Mod(a, b : string): string;
  var r: string;
  begin debug\_num(WriteLn(infofile, `\_Mod(`, a, `, `, b, `)`));
  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);
  debug\_num(WriteLn(infofile, `End_{\sqcup}Mod: `,r));
  end;
234. 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.
  define assign\_qcd\_and\_jump(\#) \equiv
            begin r \leftarrow \#; goto ex; end
\langle Basic arithmetic operations declarations 204\rangle + \equiv
function GCD(a, b : string): string; {*Note: always returns a positive value}
235. \langle Arithmetic for arbitrary-precision integers 211 \rangle + \equiv
function GCD(a, b : string): string;
  label ex;
  var a1, b1, p, r: string;
  begin a1 \leftarrow a; b1 \leftarrow b;
  debug_num(WriteLn(infofile, `GCD(`, a1, `, `, b1, `)`));
  checkzero(a1,b1); a1 \leftarrow abs(a1); b1 \leftarrow abs(b1);
  if (a1 = 1) \lor (b1 = 1) then assign\_gcd\_and\_jump(1);
  if (a1 = `0`) \land (b1 \neq `0`) then assign\_gcd\_and\_jump(b1);
  if (b1 = `0`) \land (a1 \neq `0`) then assign\_gcd\_and\_jump(a1);
  if a1 = b1 then assign\_gcd\_and\_jump(a1);
  while gt(b1, 0) do \{0 < b1\}
     begin p \leftarrow b1; b1 \leftarrow Mod(a1, b1); a1 \leftarrow p end;
  r \leftarrow a1;
ex: GCD \leftarrow r;
  debug\_num(WriteLn(infofile, `End\_GCD: `, r));
  end;
       Least common multiple. We recall lcm(a, b) = |ab|/gcd(|a|, |b|).
236.
\langle Basic arithmetic operations declarations 204\rangle + \equiv
```

function LCM(a, b: string): string; {*Note: always returns a positive value}

```
237.
        \langle Arithmetic for arbitrary-precision integers 211\rangle + \equiv
function LCM(a, b : string): string;
  var a1, b1, r: string;
  begin a1 \leftarrow a; b1 \leftarrow b;
  debug_num(WriteLn(infofile, `LCM(`, a1, `, `, b1, `)`));
  checkzero(a1,b1); a1 \leftarrow abs(a1); b1 \leftarrow abs(b1); r \leftarrow DivA(Mul(a1,b1), GCD(a1,b1)); LCM \leftarrow r;
  debug\_num(WriteLn(infofile, `End_{\sqcup}LCM: `, r));
  end:
238. 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 > 0 and b > 0, a + b is computed using Add.
\langle Basic arithmetic operations declarations 204\rangle + \equiv
function Add(a, b : string): string;
239. \langle Arithmetic for arbitrary-precision integers 211 \rangle + \equiv
function Add(a, b : string): string;
  label ex;
  var r: string;
  begin debug_num(WriteLn(infofile, `Add(`,a, `, `,b, `) `));
  checkzero(a,b);
  if (a[1] = \hat{} - \hat{}) \wedge (b[1] = \hat{} - \hat{}) then \langle Add two negative integers, and goto ex 240\rangle;
  if (a[1] \neq `-`) \land (b[1] \neq `-`) then
     begin r \leftarrow Add(a, b); goto ex; end;
  if (a[1] = (-1) \land (b[1] \neq (-1)) then \langle Calculate(-a) + b = b - a \text{ and } \mathbf{goto} \ ex \ 241 \rangle;
  if (a[1] \neq -) \land (b[1] = -) then (Calculate a + (-b) = a - b and goto ex = 242);
ex: Add \leftarrow r:
  debug\_num(WriteLn(infofile, `End\_Add: `, r));
  end:
240. \langle Add two negative integers, and goto ex 240 \rangle \equiv
  begin r \leftarrow - + Add(abs(a), abs(b));
  if r = \text{`-0'} then r \leftarrow \text{`0'};
  goto ex;
  end
This code is used in section 239.
241. \langle \text{Calculate } (-a) + b = b - a \text{ and } \mathbf{goto} \ ex \ 241 \rangle \equiv
  if qt(abs(a),b) then
     begin r \leftarrow `-` + \_Sub(abs(a), b);
     if r = -0 then r \leftarrow 0;
     goto ex;
     end
  else begin r \leftarrow \_Sub(abs(a), b); goto ex; end
This code is used in section 239.
```

```
242. \langle \text{Calculate } a + (-b) = a - b \text{ and } \mathbf{goto} \ ex \ 242 \rangle \equiv
  if qt(abs(b), a) then
     begin r \leftarrow \text{`-'} + \text{\_}Sub(abs(b), a);
     if r = \text{`-0'} then r \leftarrow \text{`0'};
     goto ex;
     end
  else begin r \leftarrow \_Sub(abs(b), a); goto ex; end
This code is used in section 239.
243. 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 sgn(x) = -1, and x \ge 0 tests if sgn(x) \ne -1.
\langle Basic arithmetic operations declarations 204\rangle + \equiv
function Sub(a, b : string): string;
244. \langle Arithmetic for arbitrary-precision integers 211 \rangle + \equiv
function Sub(a, b : string): string;
  label ex;
  var r: string;
  begin debug_num(WriteLn(infofile, `Sub(`,a,`,`,b,`)`));
  checkzero(a,b);
  if (a[1] = (-1) \land (b[1] \neq (-1)) then \langle \text{Calculate } (-a) - b = (a+b) \text{ and } \text{goto } ex \text{ 245} \rangle;
  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 \langle \text{Calculate } (-a) - (-b) \text{ and } \text{goto } ex \text{ 246} \rangle;
  if (a[1] \neq \hat{} - \hat{}) \wedge (b[1] \neq \hat{} - \hat{}) then \langle Calculate difference of two positive integers 247\rangle;
ex: Sub \leftarrow r;
  debug\_num(WriteLn(infofile, `End\_Sub: `, r));
245. \langle \text{Calculate } (-a) - b = -(a+b) \text{ and } \mathbf{goto} \ ex \ 245 \rangle \equiv
  begin r \leftarrow `-` + \_Add(abs(a), b);
  if r = -0 then r \leftarrow 0;
```

This code is used in section 244.

```
246.
        \langle \text{Calculate } (-a) - (-b) \text{ and } \mathbf{goto} \ ex \ 246 \rangle \equiv
  if gt(abs(a), abs(b)) then
     begin r \leftarrow \text{`-'} + \text{\_}Sub(abs(a), abs(b));
     if r = \text{`-0'} then r \leftarrow \text{`0'};
     goto ex;
     end
  else begin r \leftarrow \_Sub(abs(a), abs(b)); goto ex; end
This code is used in section 244.
247. \langle Calculate difference of two positive integers 247 \rangle \equiv
  if qt(b,a) then
     begin r \leftarrow \text{`-'} + \text{\_}Sub(b, a);
     if r = -0 then r \leftarrow 0;
     goto ex;
     end
  else begin r \leftarrow \_Sub(a, b); goto ex; end
This code is used in section 244.
248. Multiplication of arbitrary-precision integers. We calculate the product of a with b by handling
the case where \operatorname{sgn}(a) \neq \operatorname{sgn}(b) as ab = -|a| \cdot |b|. Otherwise we can just rely on the Mul(a,b) to do our
work.
\langle Basic arithmetic operations declarations 204\rangle + \equiv
function Mul(a, b : string): string;
249. \langle Arithmetic for arbitrary-precision integers 211 \rangle + \equiv
function Mul(a, b : string): string;
  label ex;
  var r: string;
  begin debug_num(WriteLn(infofile, `Mul(`,a,`,`,b,`)`));
  checkzero(a,b);
  if ((a[1] = \ \ \ \ ) \land (b[1] \neq \ \ \ \ )) \lor ((a[1] \neq \ \ \ \ ) \land (b[1] = \ \ \ \ )) then
     begin r \leftarrow \text{`-'} + 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;
  debug\_num(WriteLn(infofile, `End\_Mul:`,r));
  end;
250. DivA. This is the division for arbitrary-precision integers. Like multiplication, we handle the case
\operatorname{sgn}(a) \neq \operatorname{sgn}(b) by computing a/b = -|a|/|b|.
\langle Basic arithmetic operations declarations 204\rangle + \equiv
function DivA(a, b : string): string;
           { *Note: divides absolute values and preserves the sign of the division }
```

```
62
       \langle Arithmetic for arbitrary-precision integers 211\rangle + \equiv
function DivA(a, b : string): string;
  label ex;
  var r: string;
  begin debug_num(WriteLn(infofile, 'DivA(',a,',',b,')'));
  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;
  else r \leftarrow Div(abs(a), abs(b));
ex: DivA \leftarrow r;
  debug\_num(WriteLn(infofile, `End_DivA:`,r));
  end:
```

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 $x_{\text{lead}}^2 \ge x$.

```
\langle Basic arithmetic operations declarations 204\rangle + \equiv
function IsPrime(a:string): Boolean;
```

```
253. \langle Arithmetic for arbitrary-precision integers 211 \rangle + \equiv
function IsPrime(a:string): 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|.
\langle Basic arithmetic operations declarations 204\rangle + \equiv
function Divides(a, b : String): boolean;
255. \langle Arithmetic for arbitrary-precision integers 211 \rangle + \equiv
function Divides(a, b : string): Boolean;
  var r: Boolean;
  begin r \leftarrow GCD(a, b) = abs(a); Divides \leftarrow r;
  end:
```

Section 9.2. ARBITRARY-PRECISION RATIONAL ARITHMETIC

256. Rational numbers are a pair of arbitrary precision integers (represented as a String). The convention is that the denominator is a *strictly positive* integer.

```
\langle Types for arbitrary-precision arithmetic 256\rangle \equiv
   Rational = \mathbf{record} \ Num, Den: string
     end:
```

See also section 276.

This code is used in section 198.

"Zero" and "one" are frequently used rational numbers, so we should define them as constants.

```
\langle Zero and units for arbitrary-precision 257\rangle \equiv
RZero: Rational = (Num : `0`; Den : `1`);
ROne: Rational = (Num : `1`; Den : `1`);
See also section 277.
```

This code is used in section 198.

258. 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 Rational arithmetic declarations 258\rangle \equiv
procedure RationalReduce(\mathbf{var}\ r: Rational);
See also sections 260, 262, 264, 266, 268, 270, 272, and 274.
This code is used in section 198.
```

259. \langle Arbitrary-precision rational arithmetic 259 $\rangle \equiv$ **procedure** $RationalReduce(\mathbf{var}\ r: Rational);$

var lGcd: String;

begin $lGcd \leftarrow qcd(r.Num, r.Den)$; $r.Num \leftarrow diva(r.Num, lGcd)$; $r.Den \leftarrow diva(r.Den, lGcd)$;

See also sections 261, 263, 265, 267, 269, 271, 273, and 275.

This code is used in section 198.

Rational addition. We recall

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

We should return the reduced form of the result.

 \langle Rational arithmetic declarations 258 $\rangle +\equiv$

function RationalAdd (**const** r1, r2: Rational): Rational;

261. \langle Arbitrary-precision rational arithmetic $259 \rangle + \equiv$ **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;

end:

262. Rational subtraction. Similar to addition, but the numerator is ad - bc. ⟨Rational arithmetic declarations 258⟩ +≡ function RationalSub(const r1, r2: Rational): Rational;
263. ⟨Arbitrary-precision rational arithmetic 259⟩ +≡ function RationalSub(const r1, r2: Rational): Rational;
var lRes: Rational;

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

 $lRes.Den \leftarrow Mul(r1.Den, r2.Den); RationalReduce(lRes); RationalSub \leftarrow lRes;$

begin $lRes.Num \leftarrow Sub(Mul(r1.Num, r2.Den), Mul(r1.Den, r2.Num));$

 $\langle \text{Rational arithmetic declarations } 258 \rangle + \equiv$ **function** RationalNeg(**const** r1: Rational): Rational;

265. ⟨Arbitrary-precision rational arithmetic 259⟩ +≡ function RationalNeg(const r1: Rational): Rational; var lRes: Rational; begin lRes.Num ← Mul(´-1´, r1.Num); lRes.Den ← r1.Den; RationalNeg ← lRes; end;

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

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

 \langle Rational arithmetic declarations 258 \rangle + \equiv **function** RationalMult(**const** r1, r2: Rational): Rational;

267. \langle Arbitrary-precision rational arithmetic 259 \rangle + \equiv function RationalMult (const r1, r2: Rational): Rational; $var\ lRes: Rational$; begin $lRes.Num \leftarrow Mul(r1.Num, r2.Num)$; $lRes.Den \leftarrow Mul(r1.Den, r2.Den)$; $RationalMult \leftarrow lRes$; end;

268. 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).

 \langle Rational arithmetic declarations 258 $\rangle + \equiv$

function RationalInv(const r: Rational): Rational;

```
\langle Arbitrary-precision rational arithmetic 259\rangle + \equiv
function RationalInv(\mathbf{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;
       Dividing rational numbers. We see that r_1/r_2 = r_1 \times (r_2^{-1}). That's the trick.
\langle Rational arithmetic declarations 258\rangle + \equiv
function RationalDiv(const r1, r2: Rational): Rational;
271. \langle Arbitrary-precision rational arithmetic 259\rangle + \equiv
function RationalDiv(const r1, r2: Rational): Rational;
  begin RationalDiv \leftarrow RationalMult(r1, RationalInv(r2));
  end;
272. 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.
\langle Rational arithmetic declarations 258\rangle + \equiv
function RationalEq(\mathbf{const}\ r1, r2: Rational): boolean;
273. \langle Arbitrary-precision rational arithmetic 259\rangle + \equiv
function RationalEq(\mathbf{const}\ r1, r2: Rational): boolean;
  begin RationalEq \leftarrow (r1.Num = r2.Num) \land (r1.Den = r2.Den);
  end:
274.
      Testing inequality of rational numbers. We have n_1/d_1 \le n_2/d_2 if n_1d_2 \le n_2d_1.
  Similarly, we have n_1/d_1 > n_2/d_1 is just the negation of n_1/d_1 \le n_2/d_2.
\langle Rational arithmetic declarations 258\rangle + \equiv
function RationalLE(\mathbf{const}\ r1, r2:\ Rational):\ boolean;
function RationalGT (const r1, r2: Rational): boolean;
275. \langle Arbitrary-precision rational arithmetic 259 \rangle + \equiv
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:
```

Section 9.3. RATIONAL COMPLEX NUMBERS

 \langle Types for arbitrary-precision arithmetic 256 $\rangle + \equiv$

 \langle Predicate declarations for arbitrary-precision arithmetic 278 $\rangle \equiv$

276. 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.

That is to say, rational complex numbers are represented by a pair of rational numbers in Cartesian form z = p + iq.

```
    RComplex = record Re, Im: Rational end;
    277. ⟨Zero and units for arbitrary-precision 257⟩ +≡
    CZero: RComplex = (Re: (Num: `0´; Den: `1´); Im: (Num: `0´; Den: `1´));
    COne: RComplex = (Re: (Num: `1´; Den: `1´); Im: (Num: `0´; Den: `1´));
    CMinusOne: RComplex = (Re: (Num: `-1´; Den: `1´); Im: (Num: `0´; Den: `1´));
    CImUnit: RComplex = (Re: (Num: `0´; Den: `1´); Im: (Num: `1´; Den: `1´));
    278. We want to know when these numbers describe integers (i.e., the imaginary par
```

278. 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).

```
function IsIntegerNumber(const z: RComplex): Boolean;
function IsNaturalNumber(const z: RComplex): Boolean;
function IsPrimeNumber(const z: RComplex): Boolean;
See also sections 280 and 282.
This code is used in section 198.
279. \langle Complex-rational arbitrary-precision arithmetic 279\rangle \equiv
function IsIntegerNumber(const z: RComplex): Boolean;
  begin IsIntegerNumber \leftarrow (z.Im.Num = `0`) \land (z.Re.Den = `1`); end;
function IsNaturalNumber(const z: RComplex): Boolean;
  begin IsNaturalNumber \leftarrow (z.Im.Num = `0`) \land (z.Re.Den = `1`) \land (geq(z.Re.Num, `0`)); end;
function IsPrimeNumber(const z: RComplex): boolean;
  begin if IsNaturalNumber(z) \land IsPrime(z.Re.Num) then IsPrimeNumber \leftarrow true
  else IsPrimeNumber \leftarrow false;
  end;
See also sections 281, 283, 285, 287, 289, 291, 293, 296, 299, 301, 303, and 305.
This code is used in section 198.
```

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

```
\langle Predicate declarations for arbitrary-precision arithmetic 278 \rangle +\equiv function AreEqComplex(\mathbf{const}\ z1, z2:\ RComplex):\ Boolean; function IsEqWithInt(\mathbf{const}\ z:\ RComplex; n:\ longint):\ Boolean;
```

```
\langle Complex-rational arbitrary-precision arithmetic 279\rangle + \equiv
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;
        "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 Rational GT being badly named holds for IsRational GT being badly
named as well.
\langle Predicate declarations for arbitrary-precision arithmetic 278\rangle + \equiv
function IsRationalLE(const z1, z2: RComplex): Boolean;
function IsRationalGT(const z1, z2: RComplex): Boolean;
      \langle Complex-rational arbitrary-precision arithmetic 279\rangle + \equiv
function IsRationalLE(\mathbf{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;
Subsection 9.3.1. Arithmetic operations
284.
       Converting integers to complex numbers. We have a function to convert an integer x \in \mathbf{Z} to
be the complex number (x/1) + i(0/1) \in \mathbb{C}.
\langle Declare public complex-valued arbitrary precision arithmetic 284\rangle \equiv
function IntToComplex(x:integer): RComplex;
See also sections 286, 288, 290, 292, 295, 298, and 300.
This code is used in section 198.
      \langle Complex-rational arbitrary-precision arithmetic 279\rangle + \equiv
function IntToComplex(x:integer): RComplex;
  var lRes: RComplex;
  begin lRes \leftarrow COne; lRes.Re.Num \leftarrow IntToStr(x); IntToComplex \leftarrow lRes;
  end;
286.
      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).
\langle Declare public complex-valued arbitrary precision arithmetic 284\rangle + \equiv
function ComplexAdd(const z1, z2: RComplex): RComplex;
      \langle Complex-rational arbitrary-precision arithmetic 279\rangle + \equiv
function ComplexAdd(const z1, z2: RComplex): RComplex;
  var lRes: RComplex;
  begin lRes.Re \leftarrow RationalAdd(z1.Re, z2.Re); lRes.Im \leftarrow RationalAdd(z1.Im, z2.Im);
  if_def (CH_REPORT) CHReport.Out_NumReq3(rqRealAdd, z1, z2, lRes); endif
  ComplexAdd \leftarrow lRes;
  end:
```

Subtracting complex numbers. We find the difference of complex numbers componentwise. \langle Declare public complex-valued arbitrary precision arithmetic 284 $\rangle + \equiv$ **function** ComplexSub(**const** z1, z2: RComplex): RComplex; **289.** \langle Complex-rational arbitrary-precision arithmetic 279 $\rangle + \equiv$ function ComplexSub(const z1, z2: RComplex): RComplex; var lRes: RComplex; **begin** $lRes.Re \leftarrow RationalSub(z1.Re, z2.Re); lRes.Im \leftarrow RationalSub(z1.Im, z2.Im);$ **if_def** (CH_REPORT) CHReport.Out_NumReq3 (rqRealDiff, z1, z2, lRes); **end_if** $ComplexSub \leftarrow lRes$: end; **290.** Negating complex numbers. We negate a complex number -z by negating its real and imaginary parts. \langle Declare public complex-valued arbitrary precision arithmetic 284 $\rangle + \equiv$ **function** ComplexNeg(**const** z: RComplex): RComplex; **291.** \langle Complex-rational arbitrary-precision arithmetic 279 $\rangle + \equiv$ **function** ComplexNeg(**const** z: RComplex): RComplex; var lRes: RComplex; **begin** $lRes.Re \leftarrow RationalNeg(z.Re); lRes.Im \leftarrow RationalNeg(z.Im);$ if_def (CH_REPORT) CHReport.Out_NumReq2(rqRealNeg, z, lRes); end_if $ComplexNeg \leftarrow lRes;$ end; **292.** Multiplying complex numbers. We use the usual formula $(x_1 + iy_1)(x_2 + iy_2) = (x_1x_2 - y_1y_2) + i(x_1y_2 + y_1x_2).$ \langle Declare public complex-valued arbitrary precision arithmetic 284 $\rangle + \equiv$ **function** ComplexMult(**const** z1, z2: RComplex): RComplex; \langle Complex-rational arbitrary-precision arithmetic 279 $\rangle + \equiv$ function ComplexMult(const z1, z2: RComplex): RComplex; var lRes: RComplex; **begin if** IsEqWithInt(z1,-1) **then** $ComplexMult \leftarrow ComplexNeq(z2)$ else if IsEqWithInt(z2,-1) then $ComplexMult \leftarrow ComplexNeg(z1)$ else (Calculate the usual multiplication of complex numbers 294); end;

This code is used in section 293.

end

 $ComplexMult \leftarrow lRes;$

294. \langle Calculate the usual multiplication of complex numbers $294 \rangle \equiv$

 $\mathbf{begin}\ lRes.Re \leftarrow RationalSub(RationalMult(z1.Re,z2.Re), RationalMult(z1.Im,z2.Im));$ $lRes.Im \leftarrow RationalAdd(RationalMult(z1.Re, z2.Im), RationalMult(z1.Im, z2.Re));$

if_def (CH_REPORT) $CHReport.Out_NumReg3(rqRealMult, z1, z2, lRes)$; end_if

295. 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.
⟨ Declare public complex-valued arbitrary precision arithmetic 284⟩ +≡
\mathbf{function} \ \mathit{ComplexDiv}(\mathbf{const} \ \mathit{z1}, \mathit{z2} \colon \mathit{RComplex}) \colon \mathit{RComplex};
296. (Complex-rational arbitrary-precision arithmetic 279) +\equiv
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 \langle Calculate quotient for nonzero divisor 297 <math>\rangle;
  ComplexDiv \leftarrow lRes;
  end;
297. \langle Calculate quotient for nonzero divisor 297 \rangle \equiv
  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);
  if_def (CH_REPORT) CHReport.Out_NumReq3 (rqRealDiv, z1, z2, lRes); end_if
  end
This code is used in section 296.
298. Inverting complex numbers. We can now calculate z^{-1} as just 1/z.
\langle Declare public complex-valued arbitrary precision arithmetic 284\rangle + \equiv
function ComplexInv(const z: RComplex): RComplex;
       \langle Complex-rational arbitrary-precision arithmetic 279\rangle + \equiv
function ComplexInv(const z: RComplex): RComplex;
  begin ComplexInv \leftarrow ComplexDiv(COne, z); end;
300. 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).
⟨ Declare public complex-valued arbitrary precision arithmetic 284⟩ +≡
function ComplexNorm(const z: RComplex): Rational;
301. (Complex-rational arbitrary-precision arithmetic 279) +\equiv
function ComplexNorm(const z: RComplex): Rational;
```

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

Mizar Parser

else $CompareIntStr \leftarrow -1$;

end;

70

Section 9.4. COMPARISON FUNCTIONS

302. 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*.

```
\langle Declare public comparison operators for arbitrary-precision numbers 302 \rangle \equiv function CompareInt(X1, X2 : longint): integer; function CompareIntStr(X1, X2 : string): integer; See also section 304. This code is used in section 198.
```

```
303. \langle \text{Complex-rational arbitrary-precision arithmetic } 279 \rangle + \equiv 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
```

304. 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. $\langle \text{Declare public comparison operators for arbitrary-precision numbers } 302 \rangle + \equiv$ **function** $CompareComplex(\mathbf{const}\ z1, z2:\ RComplex):\ integer;$

```
305. ⟨Complex-rational arbitrary-precision arithmetic 279⟩ +≡
function CompareComplex(const z1, z2: RComplex): integer;
var lInt: integer;
begin lInt ← CompareIntStr(z1.Re.Num, z2.Re.Num);
if lInt ≠ 0 then
begin CompareComplex ← lInt; exit end;
lInt ← CompareIntStr(z1.Re.Den, z2.Re.Den);
if lInt ≠ 0 then
begin CompareComplex ← lInt; exit end;
lInt ← CompareIntStr(z1.Im.Num, z2.Im.Num);
if lInt ≠ 0 then
begin CompareComplex ← lInt; exit end;
CompareComplex ← lInt; exit end;
CompareComplex ← lInt; exit end;
CompareComplex ← CompareIntStr(z1.Im.Den, z2.Im.Den);
end;
```

Mizar Objects and Data Structures

306. 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).

We should remind the reader PASCAL has "typed pointers", meaning an object with type $\uparrow Foo$ is a pointer to a Foo object. We lookup the object for a pointer p: $\uparrow Foo$ by dereferencing it as $p\uparrow$. If foo: Foo is an instance, we can have p point to it by writing $p \leftarrow @foo$.

Further, it is idiomatic PASCAL to have for each type Foo a pointer type $PFoo = \uparrow Foo$.

307. We will refer to "some data allocated in memory" as an "Object". Alexander Stepanov and Paul McJones's *Elements of Programming* (elementsofprogramming.com) discuss object-oriented programming from a rather baroque philosophical perspective, which the reader may find enjoyable.

```
⟨ mobjects.pas 307⟩ ≡
  ⟨GNU License 4⟩
unit mobjects;
interface
uses numbers;
  ⟨Public interface for mobjects.pas 310⟩
implementation
mdebug uses info; end_mdebug
  ⟨Implementation for mobjects.pas 308⟩
end .
```

308. 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 (§430).

```
⟨ Implementation for mobjects.pas 308⟩ ≡
procedure Abstract1;
  begin RunError(211);
  end;
See also section 309.
This code is used in section 307.
```

309. The "roadmap" for the data structures implemented in this library may be summed up loosely as: we introduce a base "object" class, then we introduce a family of collections, then we conclude with classes for sequences and partial functions.

```
\langle Implementation for mobjects.pas 308 \rangle + \equiv
  \langle MObject \text{ implementation } 313 \rangle
   MStrObj implementation 318
   MList implementation 324 \{ start of collections classes \}
   MCollection implementation 350
   MExtList implementation 371 \rangle
   MSortedList implementation 388
   MSortedExtList implementation 410 \rangle
   MSortedStrList implementation 424 \rangle
   MSortedCollection implementation 429
   String collection implementation 438
   MIntCollection implementation 442
   Stacked object implementation 450
   String list implementation 454
   Int relation implementation 498 \ { start of partial functions }
   (Partial integer function implementation 507)
  \langle NatFunc \text{ implementation } 529 \rangle
   NatSeq implementation 548
   IntSequence implementation 553
   IntSet Implementation 569
  (Partial Binary integer Functions 580)
  (Partial integers to Pair of integers Functions 598)
```

310. Constant parameters. Note that the SizeOf(Pointer) is 4 on 32-bit machines, and 8 on 64-bit machines. Since

```
2\,000\,000 = 250\,000 \times 8
```

this means that a collection can have at most 250,000 items on a 64-bit machine (whereas on a 32-bit machine they can have twice that). Perhaps a better approach would be to fix MaxSize to a fixed value, then MaxSize would be assigned MaxCollectionSize * SizeOf(Pointer).

```
\langle \text{Public interface for mobjects.pas } 310 \rangle \equiv
const
          { Maximum MCollection size }
  MaxSize = 2000000;
  MaxCollectionSize = MaxSize  div SizeOf(Pointer);
  MaxListSize = MaxSize \ div \ (SizeOf(Pointer) * 2); \ \{ Maximum \ MStringList \ size \}
  MaxIntegerListSize = MaxSize \ div \ (SizeOf (integer)); \ \{ Maximum \ IntegerList \ size \}
     { MCollection error codes }
  coIndexError = -1; { Index out of range }
  coOverflow = -2; \{ Overflow \}
  coConsistentError = -3;
  coDuplicate = -5; \{ Duplicate \}
  coSortedListError = -6;
  coIndexExtError = -7;
See also sections 311, 312, 317, 321, 322, 349, 370, 387, 409, 423, 428, 437, 441, 449, 451, 452, 485, 486, 497, 506, 528, 547, 552,
     568, 579, 597, and 611.
This code is used in section 307.
```

311. Type aliases.

```
 \begin{array}{ll} \left\langle \text{Public interface for mobjects.pas } 310 \right\rangle + \equiv \\ \textbf{type} & \left\{ \text{String pointers} \right\} \\ PString = \uparrow ShortString; & \left\{ ShortString = String [255] \right\} \\ & \left\{ \text{Character set type} \right\} \\ PCharSet = \uparrow TCharSet; \\ TCharSet = \textbf{set of } char; \\ & \left\{ \text{General arrays} \right\} \\ PByteArray = \uparrow TByteArray; \\ TByteArray = \textbf{array} \left[ 0 \ldots 32767 \right] \textbf{ of } byte; & \left\{ 32767 = 2^{15} - 1 \right\} \\ PWordArray = \uparrow TWordArray; \\ TWordArray = \textbf{array} \left[ 0 \ldots 16383 \right] \textbf{ of } word; & \left\{ 16383 = 2^{14} - 1 \right\} \\ \end{array}
```

74 BASE OBJECT Mizar Parser §312

Section 10.1. BASE OBJECT

312. 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.

```
⟨ Public interface for mobjects.pas 310⟩ +≡
   { MObject base object }
   PObject = ↑MObject;
   ObjectPtr = PObject;
   MObject = object
    constructor Init;
   procedure Free; { unused }
   destructor Done; virtual;
   function CopyObject: PObject;
   function MCopy: PObject; virtual;
   end;
```

313. Note that the VER70 conditional compilation only plays a role here, in the constructor MObject.Init. And nowhere else.

The constructor will initialize the memory allocated for the *MObject* to be zero. This is true when *VER70* is not defined, too, because the Free PASCAL compiler will allocate 1 word for the virtual methods table and 1 word for the data ("self") and the default constructor ("fpc_help_constructor") for Free PASCAL initializes the memory allocated with zeros.

```
⟨ MObject implementation 313⟩ ≡
    { MObject }
constructor MObject.Init;
    @{@&$IFDEF VER70 @}
    type Image = record Link: word;
        Data: record
        end;
    end;
    end;
    @{@&$ENDIF @}
    begin
    @{@&$IFDEF VER70 @}FillChar(Image(Self).Data, SizeOf(Self) - SizeOf(MObject), 0);
    @{@&$ENDIF @}
    end;
This code is used in section 309.
```

314. Destructor. The destructor is, well, what C++ programmers would call an "abstract method".

The MObject.Free procdure frees all the memory allocated to the caller. It isn't used anywhere.

```
procedure MObject.Free; { unused }
  begin Dispose(PObject(@Self), Done); end;
destructor MObject.Done;
  begin end;
```

§315 Mizar Parser BASE OBJECT 75

315. Copying an object allocates new memory using the Free PASCAL *GetMem* function, then *copies* the contents of the caller to the new region. The *move* primitive function is poorly named (blame Borland): it is a copy function.

It then returns a pointer to the newly allocated object.

Note that this function is used in only two places: once in MCopy, and later in MList.MCopy (§329).

```
function MObject.CopyObject: PObject;

var lObject: PObject;

begin GetMem(lObject, SizeOf(Self));

Move(Self, lObject \uparrow, SizeOf(Self));

CopyObject \leftarrow lObject;

end;
```

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

```
\begin{array}{l} \textbf{function} \ \textit{MObject}.\textit{MCopy} \colon \textit{PObject}; \\ \textbf{begin} \ \textit{MCopy} \leftarrow \textit{CopyObject}; \ \textbf{end}; \end{array}
```

76 MIZAR STRING OBJECT Mizar Parser $\S317$

Section 10.2. MIZAR STRING OBJECT

317. We want to treat strings as MObjects, and the way object-oriented programming handles this situation is to create a subclass consisting of just a string field. This amounts to a "wrapper class".

```
⟨ Public interface for mobjects.pas 310⟩ +≡
    { Specyfic objects based on MObjects for collections }
    PStr = ↑MStrObj;
    MStrPtr = PStr;
    MStrObj = object (MObject)
    fStr: string;
    constructor Init(const aStr: string);
    end;

318. Constructor. The constructor for a string object expects a string, and simply initializes its contents to the given string.
⟨ MStrObj implementation 318⟩ ≡
    { Specyfic objects based on MObjects for collections }
constructor MStrObj.Init(const aStr: string);
    begin fStr ← aStr; end;
This code is used in section 309.
```

§319 Mizar Parser MIZAR LIST 77

Section 10.3. MIZAR LIST

319. 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.

320. 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 $p\uparrow \leftarrow newValue$.

```
321. ⟨Public interface for mobjects.pas 310⟩ +≡
{MCollection types}

PItemList = ↑MItemList;

MItemList = array [0...MaxCollectionSize - 1] of Pointer;
```

78 MIZAR LIST Mizar Parser $\S 322$

322. A *MList* object is known as a dynamic array. Java programmers would know that as an ArrayList. $\langle \text{Public interface for mobjects.pas } 310 \rangle + \equiv$ { 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;

323. It's worth pointing out that an *MList* does not "own" the items in it, in the sense that: when we delete an *MList* instance, we do not need to delete each item in it.

324.	Growth factor.	How quickly	an Dynamic	Array	grows is	a subject	of debate.	Just for a table	of
the gre	owth 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/

The reader wondering what strategy Free PASCAL uses should consult §8.4.1 of the "Free PASCAL Programmer's Guide" (eprint).

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,

$$\alpha^2 S \le S + \alpha S$$

which means

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

or (requiring $\alpha > 0$)

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

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

 $\langle MList \text{ implementation } 324 \rangle \equiv \{ \text{ Simple Collection } \}$

function GrowLimit(aLimit:integer): integer;

begin $GrowLimit \leftarrow 4$;

if aLimit > 64 then $GrowLimit \leftarrow aLimit$ div 4

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

end;

This code is used in section 309.

80 MIZAR LIST Mizar Parser §325

325. Constructor. The constructor creates an empty list. The initial capacity and initial size are both set to zero.

```
constructor MList.Init(aLimit:integer);
begin MObject.Init; Items \leftarrow nil; Count \leftarrow 0; Limit \leftarrow 0; SetLimit(aLimit); end;
```

326. Moving a list into the caller. Since an MList does not own its contents, moving its contents around amounts to updating pointers. The DeleteAll (§338) method just updates the capacity of the caller to zero, it does not free anything from memory.

```
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:
```

327. Copying the contents of *aAnother* list into the current list will essentially reinitialize the current list, the insert all items from the other list into the current list using InsertList (§334).

```
constructor MList.CopyList(var aAnother: MList); begin MObject.Init; Items \leftarrow nil; Count \leftarrow 0; Limit \leftarrow 0; { initialize } SetLimit(aAnother.Limit); InsertList(aAnother); end;
```

328. 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;
```

329. We override the MObject.MCopy method (§316). This will copy the base object using CopyObject (§315), allocate a new array of pointers, copy over the contents of the caller, and then returns the new list. Importantly, this will allocate new objects on the heap, duplicating every entry in the caller and the caller's data (capacity and size).

```
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;
```

330. This is the same as MList.GetObject (§335), 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;
```

331. 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;
```

§332 Mizar Parser MIZAR LIST 81

332. 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)); { ensure capacity }

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

end;
```

333. 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

PUZZLE: What happens if the user calls AtInsert(caller.Count-n, object)? The code will set every pointer in Items[Caller.Count-n . Caller.Count-1] to nil, which seems buggy.

Solution: The SetLimit method does not update the Count field of the caller, so the problem just stated will never happen.

```
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;
```

334. 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.

Observe that this has, for each item in the argument supplied, the caller *Insert* a pointer to a copy of each item. That is to say, the caller *pushes* a new item to the end of the caller's contents.

```
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};
```

335. 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;
```

82 MIZAR LIST Mizar Parser §336

336. We have a default error code for lists.

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

337. 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.

Note: this uses pointer comparison, so it will not compare the *contents* of the object for equality.

```
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;
```

338. Deleting all items from a list simply updates the list's logical size (i.e., *Count*) to zero. Important contracts which hold about this:

- This will not alter the underlying array allocated for the dynamic array.
- This will not free any allocated objects from memory.

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

339. 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(nil), Done) which would throw errors.

[This method appears to be used only by subclasses of *MList*, so I think this should be a protected method.]

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

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

```
procedure MList.FreeAll;
begin FreeItemsFrom(0); end;
```

341. 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;
```

§342 Mizar Parser MIZAR LIST 83

342. 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;
```

- **343.** 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 \langle \text{Ensure } Count \leq ALimit \leq MaxCollectionSize 344 \rangle;
  if ALimit \neq Limit then
     begin if ALimit = 0 then lItems \leftarrow nil
     else (Allocate a new array, and copy old contents into new array 345);
     if Limit \neq 0 then FreeMem(Items, Limit * SizeOf(Pointer));
     Items \leftarrow lItems; Limit \leftarrow ALimit;
     end;
  end;
        \langle \text{Ensure } Count \leq ALimit \leq MaxCollectionSize 344 \rangle \equiv
  if ALimit < Count then ALimit \leftarrow Count;
  if ALimit > MaxCollectionSize then ALimit \leftarrow MaxCollectionSize
This code is used in sections 343 and 419.
345. \langle Allocate a new array, and copy old contents into new array 345 \rangle \equiv
  begin GetMem(lItems, ALimit * SizeOf(Pointer));
  if ((Count) \neq 0) \land (Items \neq nil) then Move(Items \uparrow, IItems \uparrow, Count * SizeOf(Pointer));
  end
This code is used in section 343.
```

84 MIZAR LIST Mizar Parser §346

346. 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;
```

347. 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 [virtual method table pointer])."

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;
```

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

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

\mathbf{var}\ i:\ integer;

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

\mathbf{end};
```

Section 10.4. MIZAR COLLECTION CLASS

349. Curiously, the "Collection" class extends the "List" class, which surprises me. This will change the growth rate from s(n+1) = s(n) + GrowLimit(s(n)) to be

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

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.

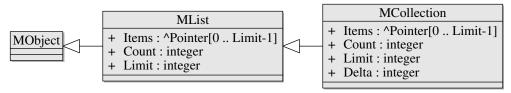


Fig. 1. UML class diagram for MCollection class.

```
\langle \text{Public interface for mobjects.pas } 310 \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;
350. Constructor. When constructing a new Collection, we allocate an array of the desired limit (using
the SetLimit (§343) to handle this allocation).
  We should have preconditions that ADelta \geq 0 and ALimit \geq 0, enforced by assertions.
\langle MCollection \text{ implementation } 350 \rangle \equiv
    { MCollection }
constructor MCollection. Init(ALimit, ADelta: integer);
  begin MObject.Init; Items \leftarrow nil; Count \leftarrow 0; Limit \leftarrow 0; Delta \leftarrow ADelta; SetLimit(ALimit);
  end:
```

See also sections 351, 352, 355, 356, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, and 369.

This code is used in section 309.

86

351. Destructor. When the destructor for an MCollection is invoked, all the items are freed using FreeAll (§340).

```
⟨ MCollection implementation 350⟩ +≡ destructor MCollection.Done; begin FreeAll; SetLimit(0); end;
```

352. Delete entry at an index. 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 to the left. This means the entry at the end of the collection (when the caller invokes this function) stays there, but the logical size of the caller decrements by one.

```
\langle MCollection \text{ implementation } 350 \rangle + \equiv
procedure MCollection.AtDelete(Index: integer);
  var i: integer;
  begin (Check Index for MCollection is within bounds, raise an error if it's out of bounds 353);
  \langle Shift entries of MCollection to the left by one 354\rangle;
  dec(Count);
  end;
       \langle Check Index for MCollection is within bounds, raise an error if it's out of bounds 353\rangle
  if (Index < 0) \lor (Index \ge Count) then
     begin ListError(coIndexError, 0); exit;
     end
This code is used in sections 352, 356, and 359.
354. \langle Shift entries of MCollection to the left by one 354\rangle \equiv
  if Index < pred(Count) then
     for i \leftarrow Index to Count - 2 do Items \uparrow [i] \leftarrow Items \uparrow [i+1]
This code is used in section 352.
```

355. Delete and free entry. 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.

```
\langle MCollection \text{ implementation } 350 \rangle +\equiv
procedure MCollection.AtFree(Index:integer);
var lItem: Pointer;
begin lItem \leftarrow At(Index); AtDelete(Index); FreeItem(lItem); end;
```

Inserting an entry at a specific location. 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

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.

```
\langle MCollection \text{ implementation } 350 \rangle + \equiv
procedure MCollection. AtInsert(Index: integer; Item: Pointer);
  begin (Check Index for MCollection is within bounds, raise an error if it's out of bounds 353);
  Ensure capacity of MCollection, raise coOverFlow error if \Delta = 0 357);
  \langle Shift entries to the right by one 358\rangle;
  Items \uparrow [Index] \leftarrow Item; inc(Count);
  end;
357. Ensure capacity of MCollection, raise coOverFlow error if \Delta = 0 357 \equiv
  if Limit = Count then { grow the caller }
     begin if Delta = 0 then
       begin ListError(coOverFlow, 0); exit;
       end;
     SetLimit(Limit + Delta);
     end
This code is used in section 356.
358. \langle Shift entries to the right by one 358\rangle \equiv
  if Index \neq Count then Move(Items \uparrow [Index], Items \uparrow [Index + 1], (Count - Index) * SizeOf(pointer))
This code is used in section 356.
359. Overwrite contents at index. We can insert a new item at a given index without shifting the
collection. This overwrites the contents of the entry at Index, which just updates the pointer's value to the
new Item.
  This can cause a memory leak, if the object located at old(Items\uparrow[Index]) is not "owned" by something
else.]]
\langle MCollection \text{ implementation } 350 \rangle + \equiv
procedure MCollection.AtPut(Index: integer; Item: Pointer);
  begin (Check Index for MCollection is within bounds, raise an error if it's out of bounds 353);
  Items \uparrow [Index] \leftarrow Item;
  end;
       Delete entry by pointer. Deleting an item finds the index of the item, then invokes AtDelete
(\S 352) on that index.
\langle MCollection implementation 350 \rangle + \equiv
procedure MCollection.Delete(Item: Pointer);
  begin AtDelete(IndexOf(Item));
  end;
```

Mizar Parser

begin SetLimit(0) **end**;

88

Free element by pointer. Similarly, freeing an item is just Delete-ing the item (§360), then calling FreeItem ($\S 339$) on the pointer. $\langle MCollection implementation 350 \rangle + \equiv$ **procedure** MCollection.Free(Item: Pointer); **begin** Delete(Item); FreeItem(Item); end; 362. Add entry at end of collection. Inserting an item at the end of the collection. $\langle MCollection \text{ implementation } 350 \rangle + \equiv$ **procedure** *MCollection.Insert(aItem: Pointer)*; **begin** AtInsert(Count, aItem); end: 363. **Delete nil pointers.** We can also "fit" the collection by deleting all **nil** elements. $\langle MCollection implementation 350 \rangle + \equiv$ **procedure** *MCollection*.*Pack*; **var** *i*: *integer*; begin for $i \leftarrow pred(Count)$ downto 0 do if $Items \uparrow [i] = nil then AtDelete(i);$ end; **364.** Move constructor. Move semantics for creating a new collection. $\langle MCollection \text{ implementation } 350 \rangle + \equiv$ **constructor** MCollection. MoveCollection(var fAnother: MCollection); **begin** Init(0, fAnother. Delta); TransferItems(fAnother) **end**; 365. Copy constructor. 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. $\langle MCollection \text{ implementation } 350 \rangle + \equiv$ **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: Singleton constructor. A singleton allocates as little as possible. $\langle MCollection implementation 350 \rangle + \equiv$ **constructor** *MCollection.Singleton(fSing : PObject; fDelta : integer)*; begin Init(2, fDelta); Insert(fSing) end; **367.** Soft delete everything. Pruning a collection merely sets its limits to zero. It does not free the contents of the collection. $\langle MCollection implementation 350 \rangle + \equiv$ **procedure** *MCollection*.*Prune*;

 $\S 368$ Mizar Parser MIZAR COLLECTION CLASS 89

368. Move list into caller. Moving an *MList* into the caller uses PASCAL's inheritance semantics to invoke *MList.MoveList* and then sets the *Delta* to 2.

```
\langle MCollection \ implementation \ 350 \rangle +\equiv
constructor MCollection.MoveList(\mathbf{var}\ aAnother: MList);
begin inherited\ MoveList(aAnother);\ Delta \leftarrow 2;
end;
```

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

```
\langle MCollection \text{ implementation } 350 \rangle + \equiv
constructor MCollection.CopyList(\mathbf{var}\ aAnother: MList);
begin inherited\ CopyList(aAnother);\ Delta \leftarrow 2; end;
```

Section 10.5. SIMPLE STACKED (EXTENDIBLE) LISTS

370. This is used to track newly registered clusters in Mizar. We want to process or "digest" entries, and track which entries have been "digested" and which ones have not. So we partition the array into the first N entries (which have been "digested") and then the remaining k "extendible" entries (which have yet to be "digested").

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

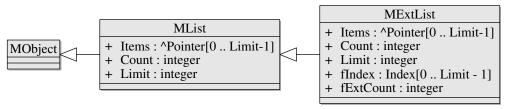


Fig. 2. UML class diagram for MExtList class.

```
\langle \text{Public interface for mobjects.pas } 310 \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(\mathbf{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;
```

371. Empty constructor. We can create an empty MExtList, which just assigns the array of items the nil value, the capacity is assigned zero, the logical size is assigned zero.

```
\langle \mathit{MExtList} \ \mathrm{implementation} \ 371 \rangle \equiv constructor \mathit{MExtList}.\mathit{Init}(\mathit{ALimit}:\mathit{integer}); begin \mathit{MObject}.\mathit{Init}; \mathit{Items} \leftarrow \mathrm{nil}; \mathit{Count} \leftarrow 0; \ \mathit{Limit} \leftarrow 0; \mathit{SetLimit}(\mathit{ALimit}); \ \mathit{fExtCount} \leftarrow 0; end; See also sections 372, 373, 375, 376, 377, 378, 379, 381, 382, 383, 384, 385, and 386. This code is used in section 309.
```

372. Destructor for MExtList. The destructor for MExtList invokes self. Free ExtItems and then calls the inherited destructor from the superclass (§328).

```
⟨ MExtList implementation 371 ⟩ +≡ destructor MExtList.Done; begin FreeExtItems; inherited Done; end;
```

373. Inserting an item. If there are "undigested" extendible items, then we have a dilemma: where is the "end" of the list where we insert the new item? Do we shift all the "undigested" extendible items to the right by 1, and insert the argument at the end of the list of "digested" items? Do we need to track segments of digested and undigested items? This is clearly problematic, so we seek a simple solution: if there are any "undigested" extendible items, raise an error.

Otherwise, we possibly grow the extendible list, and we insert at the end the given pointer and increment the *Count* of items allocated. The new item is considered "digested".

```
\langle \mathit{MExtList} \text{ implementation } 371 \rangle + \equiv
\mathbf{procedure} \ \mathit{MExtList.Insert}(\mathit{altem} : \mathit{Pointer});
\mathbf{begin} \ \langle \mathit{Check} \ \mathit{all} \ \mathit{extendible} \ \mathit{items} \ \mathit{have} \ \mathit{been} \ \mathit{digested}, \ \mathit{otherwise} \ \mathit{raise} \ \mathit{error} \ 374 \rangle;
\mathbf{if} \ \mathit{Limit} = \mathit{Count} \ \mathbf{then} \ \mathit{SetLimit}(\mathit{Limit} + \mathit{GrowLimit}(\mathit{Limit}));
\mathit{Items} \uparrow [\mathit{Count}] \leftarrow \mathit{aItem}; \quad \{ \ \mathit{Append} \ \mathit{the} \ \mathit{item} \ \mathit{to} \ \mathit{the} \ \mathit{list} \}
\mathit{inc}(\mathit{Count});
\mathbf{end};
```

374. Many methods for the MExtList class only make sense when all the extendible entries have been "digested" into the underlying array (i.e., when fExtCount = 0), otherwise we end up in a quagmire like the one outlined when discussing the Insert method and nothing makes sense anymore.

```
⟨ Check all extendible items have been digested, otherwise raise error 374⟩ ≡ if fExtCount ≠ 0 then begin ListError(coIndexExtError, 0); exit; end
This code is used in sections 373, 375, 376, 377, 382, and 416.
```

375. Deleting all entries. When all the extendible entries have been "digested", we call the parent's *DeleteAll* method.

```
⟨ MExtList implementation 371 ⟩ +≡
procedure MExtList.DeleteAll;
begin ⟨ Check all extendible items have been digested, otherwise raise error 374 ⟩;
inherited DeleteAll; { (§338) }
end;
```

376. 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).

```
\langle \mathit{MExtList} \text{ implementation } 371 \rangle + \equiv
procedure \mathit{MExtList.FreeAll};
begin \langle \mathsf{Check} \text{ all extendible items have been digested, otherwise raise error } 374 \rangle;
\mathit{inheritedFreeAll}; \quad \{ (\S 340) \}
\mathsf{end};
```

377. 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.

```
\langle \mathit{MExtList} \text{ implementation } 371 \rangle +\equiv
procedure \mathit{MExtList.Pack};
begin \langle \mathsf{Check} \text{ all extendible items have been digested, otherwise raise error } 374 \rangle;
\mathit{inheritedPack}; \quad \{ (\S 342) \}
\mathit{end};
```

378. 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 $k \leftarrow k + 1$.

```
\langle \mathit{MExtList} \text{ implementation } 371 \rangle + \equiv
procedure \mathit{MExtList}.\mathit{InsertExt}(\mathit{AItem} : \mathit{Pointer});
begin if \mathit{Limit} = \mathit{Count} + \mathit{fExtCount} then \mathit{SetLimit}(\mathit{Limit} + \mathit{GrowLimit}(\mathit{Limit}));
\mathit{Items} \uparrow [\mathit{Count} + \mathit{fExtCount}] \leftarrow \mathit{AItem}; \; \mathit{inc}(\mathit{fExtCount});
end;
```

379. 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.

```
\langle MExtList \text{ implementation } 371 \rangle + \equiv
procedure MExtList.SetLimit(ALimit: integer);
  var lItems: PItemList;
  begin \langle \text{Ensure } Count + fExtCount \leq ALimit \leq MaxCollectionSize 380 \rangle;
  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, IItems\uparrow, (Count + fExtCount) * SizeOf(Pointer));
       end;
     if Limit \neq 0 then FreeMem(Items, Limit * SizeOf(Pointer));
     Items \leftarrow lItems; \ Limit \leftarrow ALimit;
     end:
  end;
380. \langle \text{Ensure } Count + fExtCount \leq ALimit \leq MaxCollectionSize 380 \rangle \equiv
  if ALimit < Count + fExtCount then ALimit \leftarrow Count + fExtCount;
  if ALimit > MaxCollectionSize then ALimit \leftarrow MaxCollectionSize
This code is used in section 379.
```

Mark the logical size. "Marking" an extendible list amounts to setting the procedure's variable to the logical size of the extendible list.

```
This does not appear to be used anywhere.
\langle MExtList \text{ implementation } 371 \rangle + \equiv
procedure MExtList.Mark(var aIndex : integer);
  begin aIndex \leftarrow Count;
  end;
```

Hard delete "digested" items from an index to end. 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*.

```
\langle MExtList \text{ implementation } 371 \rangle + \equiv
procedure MExtList.FreeItemsFrom(aIndex: integer);
  var I: integer;
  begin (Check all extendible items have been digested, otherwise raise error 374);
  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;
```

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 $k \leftarrow k-1$.

```
\langle MExtList \text{ implementation } 371 \rangle + \equiv
procedure MExtList.AddExtObject;
  begin if fExtCount \leq 0 then
     begin ListError(coIndexExtError, 0); exit;
  inc(Count); dec(fExtCount);
  end;
```

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).

```
\langle MExtList \text{ implementation } 371 \rangle + \equiv
procedure MExtList.AddExtItems;
  begin Count \leftarrow Count + fExtCount; fExtCount \leftarrow 0;
  end;
```

Soft delete all extendible items. 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.

```
\langle MExtList \text{ implementation } 371 \rangle + \equiv
procedure MExtList.DeleteExtItems;
   begin fExtCount \leftarrow 0;
   end:
```

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

§387 Mizar Parser SORTED LISTS 95

Section 10.6. SORTED LISTS

387. 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, ..., Length(Items) - 2, we have Items \uparrow [fIndex \uparrow [i]] \leq 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.

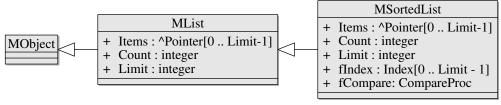


Fig. 3. UML class diagram for MSortedList class.

```
\langle \text{Public interface for mobjects.pas } 310 \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;
```

96 SORTED LISTS Mizar Parser §388

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. This will also empty the other list. (4) CopyList is like MoveList but leaves the other list untouched. $\langle MSortedList \text{ implementation } 388 \rangle \equiv$ { 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 sections 389, 390, 391, 392, 394, 398, 400, 401, 404, 405, 407, and 408. This code is used in section 309. Move constructor. When we move items from an MList into the caller. A new array will be allocated for the indices. No sorting will occur. But the argument will be mutated to delete all the elements. $\langle MSortedList \text{ implementation } 388 \rangle + \equiv$ 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; The CopyList constructor is like the MoveList except that the other list is not modified, and a new array will be allocated for the *Items*. $\langle MSortedList \text{ implementation } 388 \rangle + \equiv$ **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;

§391 Mizar Parser SORTED LISTS 97

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

```
\langle \mathit{MSortedList} \ \mathsf{implementation} \ 388 \rangle + \equiv \{ \ \mathsf{used} \ \mathsf{in} \ \mathsf{CollectCluster} \ \mathsf{not} \ \mathsf{to} \ \mathsf{repeat} \ \mathsf{the} \ \mathsf{search}, \ \mathsf{should} \ \mathsf{be} \ \mathsf{used} \ \mathsf{only} \ \mathsf{when} \ @\mathit{fCompare} \neq \mathbf{nil} \ \} 
\mathbf{procedure} \ \mathit{MSortedList}. \ \mathit{AtInsert} \ (\mathit{aIndex} : \mathit{integer}; \ \mathit{aItem} : \mathit{Pointer});
\mathbf{begin} \ \mathsf{if} \ \mathit{Limit} = \mathit{Count} \ \mathsf{then} \ \ \mathit{SetLimit} \ (\mathit{Limit} + \mathit{GrowLimit} \ (\mathit{Limit})); \ \ \{ \ \mathsf{Ensure} \ \mathsf{capacity} \ \} 
\mathbf{if} \ \mathit{aIndex} \neq \mathit{Count} \ \mathsf{then} 
\mathit{Move} \ (\mathit{fIndex} \uparrow [\mathit{aIndex} \uparrow [\mathit{aIndex} + 1], (\mathit{Count} - \mathit{aIndex}) * \mathit{SizeOf} \ (\mathit{integer})); 
\mathit{Items} \uparrow [\mathit{Count}] \leftarrow \mathit{aItem}; \ \mathit{fIndex} \uparrow [\mathit{aIndex}] \leftarrow \mathit{Count}; \ \mathit{inc} \ (\mathit{Count}); 
\mathbf{end};
```

- **392.** 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 (§401), which will mutate the Undex to be where it should be located. When the item is missing, simply insert it at Undex. 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.

```
⟨ MSortedList implementation 388⟩ +≡
procedure MSortedList.Insert(aItem : Pointer);
var Undex: integer;
begin if @fCompare = nil then ⟨Insert item to the end of the caller 393⟩;
if ¬Find(aItem, Undex) then AtInsert(Undex, aItem);
end;

393. ⟨Insert item to the end of the caller 393⟩ ≡
begin if Limit = Count then SetLimit(Limit + GrowLimit(Limit));
Items↑[Count] ← aItem; fIndex↑[Count] ← Count; inc(Count); exit;
end

This code is used in section 392.
```

394. 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).

```
⟨MSortedList implementation 388⟩ +≡
procedure MSortedList.SetLimit(aLimit:integer);
var Utems: PItemList; Undex: IndexListPtr;
begin ⟨Ensure Count ≤ aLimit ≤ MaxCollectionSize for sorted lists 395⟩;
if aLimit ≠ Limit then
begin if aLimit = 0 then
begin IItems ← nil; Undex ← nil; end
else ⟨Allocate and copy items to new sorted list 396⟩;
⟨Free old arrays, if any 397⟩;
Items ← Utems; fIndex ← Undex; Limit ← aLimit;
end;
end;
395. ⟨Ensure Count ≤ aLimit ≤ MaxCollectionSize for sorted lists 395⟩ ≡
if aLimit < Count then aLimit ← Count;
if aLimit > MaxCollectionSize then aLimit ← MaxCollectionSize
```

This code is used in section 394.

98 SORTED LISTS Mizar Parser §396

```
396. ⟨Allocate and copy items to new sorted list 396⟩ ≡
begin GetMem(Ultems, aLimit * SizeOf (Pointer)); GetMem(Undex, aLimit * SizeOf (integer));
if Count ≠ 0 then
begin if Items ≠ nil then
begin Move(Items↑, Ultems↑, Count * SizeOf (Pointer));
Move(fIndex↑, Undex↑, Count * SizeOf (integer));
end;
end;
end
This code is used in sections 394 and 419.
397. ⟨Free old arrays, if any 397⟩ ≡
if Limit ≠ 0 then
begin FreeMem(Items, Limit * SizeOf (Pointer)); FreeMem(fIndex, Limit * SizeOf (integer));
end
This code is used in sections 394 and 419.
```

§398 Mizar Parser SORTED LISTS 99

398. Quick sort an array. We have a private helper function for quicksorting an IndexListPtr (§387). Initially $L \leftarrow 0$ and $R \leftarrow length(aList) - 1$. Specifically Mizar appears to use Hoare partitioning.

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 $P \leftarrow aList \uparrow [aIndex \uparrow [(L+R)$ **shr** 1]]. Observe $I \leq P_{idx} \leq J$ at this point.
- **S1.** [Move I right] While aList[aIndex[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[aIndex[I]] = P. (Invariant: for all $0 \le h < I$, we have $aList\uparrow[aIndex\uparrow[h]] < P$.)
- **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. (Invariant: for all $J < k \leq Count 1$, we have $P < aList \uparrow [aIndex[k]]$.)
- 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 $I \ge R$, 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
\langle MSortedList \text{ implementation } 388 \rangle + \equiv
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]];
         \{I \leq (L+R) \operatorname{\mathbf{shr}} 1 \leq 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 \leq (L+R) \operatorname{shr} 1 \leq 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 < (L+R) \text{ shr } 2 < 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;

399. Remarks.

(1) It is unclear to me whether we must have *aCompare* 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) It may be instructive to compare this quicksort to Algorithm Q in *The Art of Computer Programming*, third ed., volume 3, §5.2.2.
- (5) 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)
- (6) 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.)
- (7) 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.

400. 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 (§398) to do the actual sorting.

 $\langle MSortedList \text{ implementation } 388 \rangle + \equiv$

procedure MSortedList.Sort(aCompare : CompareProc);

var *I*: *integer*;

begin $fCompare \leftarrow aCompare;$

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

 $if \ (\textit{Count} > 0) \ then \ \textit{ListQuickSort}(\textit{Items}, \textit{fIndex}, 0, \textit{Count} - 1, a\textit{Compare}); \\$

end;

§401 Mizar Parser SORTED LISTS 101

401. 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).

```
\langle \mathit{MSortedList} \text{ implementation } 388 \rangle +\equiv function \mathit{MSortedList.Find}(\mathit{aKey}:\mathit{Pointer}; \mathbf{var} \mathit{aIndex}:\mathit{integer}): \mathit{Boolean}; \mathbf{var} \ \mathit{L}, \mathit{H}, \mathit{I}, \mathit{C}: \mathit{integer}; \mathbf{begin} \ \mathit{Find} \leftarrow \mathit{False}; \mathbf{if} \ @\mathit{fCompare} = \mathbf{nil} \ \mathbf{then} \ \langle \mathit{Find} \ \mathit{needle} \ \mathit{in} \ \mathit{MSortedList} \ \mathit{by} \ \mathit{brute} \ \mathit{force} \ 403 \rangle; \langle \mathit{Find} \ \mathit{needle} \ \mathit{in} \ \mathit{MSortedList} \ \mathit{by} \ \mathit{binary} \ \mathit{search} \ 402 \rangle \mathbf{end}:
```

402. 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 402 \rangle \equiv L \leftarrow 0; H \leftarrow Count - 1; while L \leq H do

begin I \leftarrow (L+H) \operatorname{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;

This code is used in section 401.
```

403. 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 403 \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 401.

102 SORTED LISTS Mizar Parser $\S404$

404. 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 MSortedList \text{ implementation } 388 \rangle +\equiv
function MSortedList.Search(aKey: Pointer; \mathbf{var} \ aIndex: integer): Boolean;
\mathbf{var} \ I: integer;
\mathbf{begin} \ aIndex \leftarrow Count; \ Search \leftarrow false;
\mathbf{if} \ Find(aKey, I) \ \mathbf{then}
\mathbf{begin} \ Search \leftarrow true; \ aIndex \leftarrow fIndex \uparrow [I];
\mathbf{end};
\mathbf{end};
```

405. 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 (§401), 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. [The assertion that fCompare is present turns out to be unnecessary: the rest of the function works.]

```
⟨ MSortedList implementation 388⟩ +≡
function MSortedList.IndexOf (aItem : Pointer): integer;
var I: integer;
begin ⟨ Assert fCompare is present 406⟩;
IndexOf ← −1;
if Find(aItem, I) then
begin { if I < fCount then }
IndexOf ← fIndex↑[I];
end;
end;
end;
end;</pre>
406. ⟨ Assert fCompare is present 406⟩ ≡
if @fCompare = nil then
begin ListError(coSortedListError, 0); exit;
end
This code is used in sections 405 and 426.
```

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

```
\langle \mathit{MSortedList} \ \mathrm{implementation} \ 388 \rangle + \equiv procedure \mathit{MSortedList.Pack}; var \mathit{lCount} : \mathit{integer}; begin \mathit{lCount} \leftarrow \mathit{Count}; \ \mathit{inheritedPack}; \ \{ (\S 342) \} if (@\mathit{fCompare} \neq \mathbf{nil}) \land (\mathit{lCount} > \mathit{Count}) then \mathit{Sort}(\mathit{fCompare}); end;
```

 $\S408$ Mizar Parser SORTED LISTS 103

408. 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.

```
 \langle \mathit{MSortedList} \ \mathsf{implementation} \ 388 \rangle + \equiv \\ \mathbf{procedure} \ \mathit{MSortedList}.\mathit{FreeItemsFrom}(\mathit{aIndex} : \mathit{integer}); \\ \mathbf{var} \ \mathit{I}, k: \ \mathit{integer}; \\ \mathbf{begin} \ \mathsf{if} \ \mathit{aIndex} = \mathit{Count} \ \mathsf{then} \ \mathit{exit}; \\ \{ \mathsf{Delete} \ \mathsf{entries} \ \mathsf{from} \ \mathsf{the} \ \mathit{array} \ \mathsf{of} \ \mathsf{values} \} \\ \mathbf{for} \ \mathit{I} \leftarrow \mathit{aIndex} \ \mathsf{to} \ \mathit{Count} - 1 \ \mathsf{do} \\ \mathbf{if} \ \mathit{Items} \uparrow [\mathit{I}] \neq \mathbf{nil} \ \mathsf{then} \ \mathit{Dispose}(\mathit{PObject}(\mathit{Items} \uparrow [\mathit{I}]), \mathit{Done}); \\ \{ \mathsf{Update} \ \mathsf{the} \ \mathit{array} \ \mathsf{of} \ \mathsf{indices} \} \\ k \leftarrow 0; \\ \mathbf{for} \ \mathit{I} \leftarrow 0 \ \mathsf{to} \ \mathit{Count} - 1 \ \mathsf{do} \\ \mathbf{begin} \ \mathsf{if} \ \mathit{Index} \uparrow [\mathit{I}] < \mathit{aIndex} \ \mathsf{then} \\ \mathbf{begin} \ \mathit{fIndex} \uparrow [\mathit{I}] < \mathit{aIndex} \ \mathsf{then} \\ \mathbf{begin} \ \mathit{fIndex} \uparrow [\mathit{I}]; \ \mathit{inc}(k); \ \mathbf{end}; \\ \mathbf{end}; \\ \mathbf{if} \ \ \mathit{k} \neq \mathit{aIndex} \ \mathsf{then} \ \ \mathit{ListError}(\mathit{coSortedListError}, 0); \\ \mathit{Count} \leftarrow \mathit{aIndex}; \\ \mathbf{end}; \\ \mathbf{end}; \\ \mathbf{end}; \\ \end{aligned}
```

Section 10.7. SORTED EXTENDIBLE LISTS

409. We want to handle a sorted ($\S387$) version of extendible lists ($\S370$) — 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.

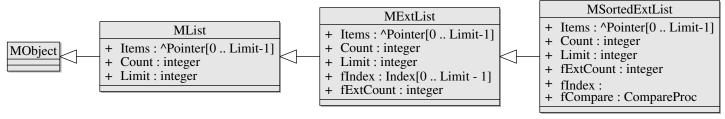


Fig. 4. UML class diagram for MSortedExtList class.

```
\langle \text{Public interface for mobjects.pas } 310 \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;
    \mathbf{procedure}\ \mathit{AddExtItems};\ \mathit{virtual};
  end;
```

410. 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 410⟩ ≡ { 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;
See also sections 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, and 422.
This code is used in section 309.
```

§411 Mizar Parser SORTED EXTENDIBLE LISTS 105

411. **Destructor** The destructor for sorted extendible lists is just the inherited destructor from extendible lists.

```
⟨ MSortedExtList implementation 410 ⟩ +≡ destructor MSortedExtList.Done; begin inherited Done; end;
```

412. 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 hay stack. Once found, we mutate aIndex to the index L of the fIndex array which indexes the needle.

The claim is that this will find the left-most index for the needle. To see this claim, work out the result of searching for "d" in the list [a, b, c, d, d, d, e, f, g] where indices range from 0 to 9, using the usual ordering a < b < c < d < e < f < g.

It is also instructive to work out the case searching for c in the list [a,b,c,c,d] indexed from 0 to 4. Again, the ordering is a < b < c < d. The first iteration will set $I \leftarrow 2$, which points to the first entry with a c. (I have written a unit test for this case, it is true: Find will return true and mutate $aIndex \leftarrow 2$.)

Ensures: if the result is true, then aIndex will be the left-most index for aKey.

```
\langle MSortedExtList \text{ implementation } 410 \rangle + \equiv
{ find the left-most if duplicates }
function MSortedExtList.Find(aKey: Pointer; var aIndex: integer): Boolean;
  var L, { low index }
     H, { high index }
     I, \{ \text{index of candidate entry } \}
     C: integer; \{ result comparing entry I to aKey \}
  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) shr 1; C \leftarrow fCompare(Items \uparrow [fIndex \uparrow [I]], aKey);
     if C < 0 then L \leftarrow I + 1 { needle is in right half of haystack }
     else begin H \leftarrow I - 1;
       if C = 0 then Find \leftarrow True;
       end;
     end;
  aIndex \leftarrow L;
  end;
```

106

413. 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.

Ensures: if the result is true, then the entry at aIndex - 1 will be equal to the aKey but the entry at aIndex will not be equal to aKey.

```
define scan\_right(\#) \equiv \text{while} \ (\# < Count) \land (0 = fCompare(Items \uparrow [fIndex \uparrow [\#]], aKey)) \ do \ inc(\#) \ \langle MSortedExtList \ implementation \ 410 \rangle + \equiv \ \{ \text{find the left-most with higher aKey, this is where we can insert } \} \ function \ MSortedExtList.FindRight(aKey: Pointer; var aIndex: integer): Boolean; begin if \ Find(aKey, aIndex) \ then \ begin \ scan\_right(aIndex); \ FindRight \ \leftarrow true; \ end \ else \ FindRight \ \leftarrow false; \ end; \end{array}
```

414. Finding all instances of a needle. 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 present, the result will have $aLeft \leq aRight$ and for any i such that $aLeft \leq i \leq aRight$ we will have $compare(Items\uparrow[fIndex\uparrow[i]], aKey) = 0$.

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

415. 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.

```
\langle MSortedExtList \text{ implementation } 410 \rangle +\equiv  function MSortedExtList.AtIndex(aIndex:integer): Pointer; begin if <math>(aIndex < 0) \lor (aIndex \ge Count) then ListError(coIndexExtError, 0); AtIndex \leftarrow Items \uparrow [fIndex \uparrow [aIndex]]; end;
```

416. 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*.

```
⟨MSortedExtList implementation 410⟩ +≡
procedure MSortedExtList.Insert(aItem: Pointer);
begin ⟨Check all extendible items have been digested, otherwise raise error 374⟩;
InsertExt(aItem); AddExtObject;
end;
```

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

```
⟨ MSortedExtList implementation 410⟩ +≡ procedure MSortedExtList.Pack; begin ListError(coIndexExtError, 0); end:
```

418. 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.

```
\langle \mathit{MSortedExtList} \text{ implementation } 410 \rangle +\equiv 
procedure \mathit{MSortedExtList}.\mathit{InsertExt}(\mathit{AItem} : \mathit{Pointer});
begin if \mathit{Limit} = \mathit{Count} + \mathit{fExtCount} then \mathit{SetLimit}(\mathit{Limit} + \mathit{GrowLimit}(\mathit{Limit}));
\mathit{Items} \uparrow [\mathit{Count} + \mathit{fExtCount}] \leftarrow \mathit{AItem}; \; \mathit{inc}(\mathit{fExtCount});
end:
```

419. 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.

```
\( \lambda MSortedExtList \) implementation 410 \rangle +\( \equiv \)

\( \text{procedure } MSortedExtList.SetLimit(ALimit : integer); \)

\( \text{var } \) \( \text{ltems: } PItemList; \) \( \text{lIndex: } IndexListPtr; \)

\( \text{begin } Count \leftarrow Count + fExtCount; \) \( \text{Ensure } Count \leq ALimit \leq MaxCollectionSize \) 344 \rangle; \( \text{if } aLimit \neq Limit \text{ then} \)

\( \text{begin } if aLimit = 0 \text{ then} \)

\( \text{begin } lItems \leftarrow \text{nil}; \) \( \text{lIndex} \leftarrow \text{nil}; \)

\( \text{end} \)

\( \text{else } \leq Allocate \text{ and } \text{copy items to new sorted list } 396 \rangle; \)

\( \text{Verten} \) \( \text{Pree old arrays, if any } 397 \rangle; \)

\( \text{Items} \leftarrow \text{lItems; } fIndex \leftarrow \text{lIndex; } Limit \leftarrow aLimit; \)

\( \text{Update the caller to use new arrays } \)

\( \text{end}; \)

\( \text{Count} \leftarrow Count - fExtCount; \)

\( \text{end}; \)
```

108

420. 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).

```
\langle MSortedExtList \text{ implementation } 410 \rangle + \equiv
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 > 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;
  if k \neq aIndex then ListError(coSortedListError, 0);
  Count \leftarrow aIndex:
  end:
```

421. 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.

```
\langle \mathit{MSortedExtList} implementation 410 \rangle +\equiv procedure \mathit{MSortedExtList}.\mathit{AddExtObject}; var \mathit{Undex}: \mathit{integer}; begin if \mathit{fExtCount} \leq 0 then \mathit{ListError}(\mathit{coIndexExtError}, 0); \mathit{FindRight}(\mathit{Items} \uparrow [\mathit{Count}], \mathit{Undex}); if \mathit{Undex} \neq \mathit{Count} then \{ \mathsf{shift} \mathit{fIndex} \mathsf{to} \; \mathsf{right} \; \mathsf{by} \; 1 \} \mathit{Move}(\mathit{fIndex} \uparrow [\mathit{Undex}], \mathit{fIndex} \uparrow [\mathit{Undex} + 1], (\mathit{Count} - \mathit{Undex}) * \mathit{SizeOf}(\mathit{integer})); \mathit{fIndex} \uparrow [\mathit{Undex}] \leftarrow \mathit{Count}; \; \{ \; \mathsf{extendible} \; \mathsf{item's} \; \mathsf{index} \; \} \mathit{inc}(\mathit{Count}); \; \mathit{dec}(\mathit{fExtCount}); end;
```

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

```
⟨ MSortedExtList implementation 410⟩ +≡
procedure MSortedExtList.AddExtItems;
begin while fExtCount > 0 do AddExtObject;
end;
```

§423 Mizar Parser SORTED LIST OF STRINGS 109

Section 10.8. SORTED LIST OF STRINGS

423. This is used in the kernel to track directives, as well as makenv and accdict needs it.

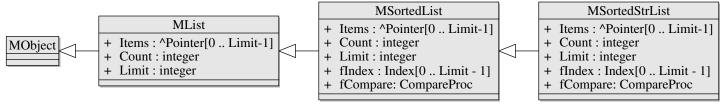


Fig. 5. UML class diagram for MSortedStrList class.

```
⟨ Public interface for mobjects.pas 310⟩ +≡
MSortedStrList = object (MSortedList)
constructor Init(ALimit : integer);
function IndexOfStr(const aStr: string): integer; virtual;
function ObjectOf(const aStr: string): PObject; virtual;
end;
```

424. 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 } 424 \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 309.
```

425. Constructor. We just defer to the *InitSorted* constructor for sorted lists (§388).

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;
```

426. We can locate a string by *Find*-ing its entry in the *fIndex* array.

```
function MSortedStrList.IndexOfStr(\mathbf{const}\ aStr:\ string): integer; var I:\ integer;\ lStringObj:\ MStrObj; begin IndexOfStr \leftarrow -1;\ \langle \operatorname{Assert}\ fCompare\ is\ present\ 406 \rangle;\ \{\operatorname{Invariant\ violation}\} lStringObj.Init(aStr); if Find(@lStringObj,I) then begin if I< Count\ then\ IndexOfStr \leftarrow fIndex\uparrow[I]; end; end;
```

427. 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;
```

110 SORTED COLLECTIONS Mizar Parser §428

Section 10.9. SORTED COLLECTIONS

428. With *MSortedList*, it was cheaper to move indices around than moving object around. But the goal was to have an array which we could sort "fast enough".

What we want to consider now is a "finite set"-like collection. The usual way to implement such a data structure is to use a sorted list, and to avoid inserting duplicate entries.

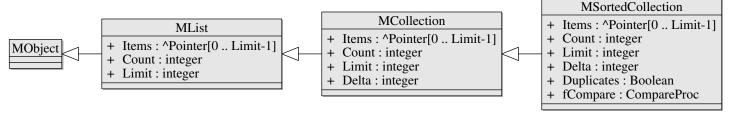


Fig. 6. UML class diagram for MSortedCollection class.

```
⟨ Public interface for mobjects.pas 310⟩ +≡
   { MSortedCollection object }
   PSortedCollection = ↑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;
```

429. Constructors. We can construct a sorted collection without an ordering operator, and we can construct one with an ordering operator.

```
\langle \mathit{MSortedCollection} \; | \;  { MSortedCollection implementation 429 \rangle \equiv  { MSortedCollection init (a\mathit{Limit}, a\mathit{Delta} : integer); constructor \mathit{MSortedCollection.Init}(a\mathit{Limit}, a\mathit{Delta} : integer); begin \mathit{inheritedInit}(A\mathit{Limit}, A\mathit{Delta}); \mathit{Duplicates} \leftarrow \mathit{False}; \mathit{fCompare} \leftarrow \mathbf{nil}; end; constructor \mathit{MSortedCollection.InitSorted(a\mathit{Limit}, a\mathit{Delta} : integer}; a\mathit{Compare} : \mathit{CompareProc}); begin \mathit{inheritedInit}(A\mathit{Limit}, A\mathit{Delta}); \mathit{Duplicates} \leftarrow \mathit{False}; \mathit{fCompare} \leftarrow \mathit{aCompare}; end; This code is used in section 309.
```

430. Comparing entries. This will invoke *Abstract1* ($\S 308$) 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;
```

§431 Mizar Parser SORTED COLLECTIONS 111

431. 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 ← Item;
end;
```

432. Binary search. This is binary search through a sorted collection. This will mutate the *Index* argument to point to where the *Key* is located if it is present (or where it should be located if it is absent). If there are duplicates, this will assign *Index* the left-most index.

When the *Key* is present in the collection, this function will return *true*. Otherwise, it will return *false*. It may be instructive for the reader to compare this to *MSortedExtList*'s *Find* method (§412).

[I think if there are no duplicates, then "begin; $Index \leftarrow L$; exit; end;" should be the body of the if statement. Nothing changes, but it avoids needless iterations.]

```
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) \text{ 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; end;
```

433. Find the right-most index for an item in the collection. Searching ($\S432$) for the KeyOf ($\S431$). I have just unit tested this, the branch incrementing I is buggy. It should be something like:

```
while (I + 1 < Count) \land (aItem = Items \uparrow [I + 1]) do inc(I); IndexOf \leftarrow I;
```

This will scan to the right until we get to the end of the collection, or the next item is no longer the same thing the user is searching for.

```
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;
```

112 SORTED COLLECTIONS Mizar Parser $\S434$

434. 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 AtInsert(I, aItem);
  end;
435.
      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;
       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;

§437 Mizar Parser STRING COLLECTION 113

Section 10.10. STRING COLLECTION

```
437. The librery pas module uses both the MStringCollection and StringColl data structures.
\langle \text{Public interface for mobjects.pas } 310 \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 438\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 309.
439. 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;
```

114 INT COLLECTIONS Mizar Parser §441

Section 10.11. INT COLLECTIONS

```
The TIntItem is needed for the unifier and equalizer.
\langle \text{Public interface for mobjects.pas } 310 \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;
       TIntItem constructor. This just copies the given integer over to the newly allocated TIntItem
442.
object.
\langle MIntCollection implementation 442 \rangle \equiv
    { MIntCollection }
constructor TIntItem.Init(fInt:integer);
  begin IntKey \leftarrow fInt;
  end;
This code is used in section 309.
443. We use TIntItems as keys in a TIntKeyCollection.
function TIntKeyCollection.KeyOf(Item: pointer): pointer;
  begin KeyOf \leftarrow addr(PIntItem(Item)\uparrow.IntKey);
  end;
```

§444 Mizar Parser INT COLLECTIONS 115

```
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;
445. Constructor for pairs of integers.
constructor IntPairItem.Init(X, Y : integer);
  begin fKey.X \leftarrow X; fKey.Y \leftarrow Y;
  end;
446. 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;
447. 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

Section 10.12. STACKED LIST OF OBJECTS

```
"Stacked" lists are really linked lists. This is an abstract class, so the methods are not implemented
(and trying to use them will raise an error).
\langle \text{Public interface for mobjects.pas } 310 \rangle + \equiv
     { Stacked Object (List of objects) }
  StackedPtr = \uparrow StackedObj;
  StackedObj = \mathbf{object} (MObject)
     Previous: StackedPtr;
     constructor Init;
     destructor Done; virtual;
  end;
       The constructors and destructors are not implemented, so if you try to use them, just raise an error.
\langle Stacked object implementation 450 \rangle \equiv
     { Stacked Object (List of objects) }
constructor StackedObj.Init;
  \mathbf{begin}\ \mathit{Abstract1}\,;
  end;
destructor StackedObj.Done;
  \mathbf{begin}\ \mathit{Abstract1}\,;
  end;
This code is used in section 309.
```

§451 Mizar Parser STRING LIST 117

Section 10.13. STRING LIST

451. We will want to use a dictionary whose keys are strings, and values are pointers to *MObjects*. The *MStringList* is such a dictionary. It's little more than an array of *MStringItem* entries, sorted according to key.

```
\langle \text{Public interface for mobjects.pas } 310 \rangle + \equiv
    { MStringList object }
  PStringItem = \uparrow MStringItem;
  MStringItem = \mathbf{record} \ fString: \ PString; \ \{ \text{key } \}
    fObject: PObject; { value }
    end;
452.
      \langle \text{Public interface for mobjects.pas } 310 \rangle + \equiv
  MDuplicates = (dupIgnore, dupAccept, dupError);
  PStringItemList = \uparrow MStringItemList;
  MStringItemList = array [0...MaxListSize] of MStringItem;
  PStringList = \uparrow MStringList;
  MStringList = \mathbf{object} \ (MObject)
    fList: PStringItemList;
    fCount: integer; { logical size }
    fCapacity: integer; { capacity of array }
    fSorted: Boolean;
    fDuplicate: MDuplicates; { how to handle duplicates }
    constructor Init(aCapacity : integer);
    constructor MoveStringList(var aAnother : MStringList);
     (Declare internal methods for StringList 453)
    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; const aStr: string; aObject: PObject);
  end;
```

118 STRING LIST Mizar Parser $\S453$

```
453.
        These methods should be declared 'private', to enforce the implicit constraint that these methods
should not be used directly.
\langle Declare internal methods for StringList 453 \rangle \equiv
     { - 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);
This code is used in section 452.
454. 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.
\langle String list implementation 454 \rangle \equiv
                      - MStringList
constructor MStringList.Init(aCapacity : integer); { empty dictionary constructor }
  begin MObject.Init; fList \leftarrow nil; fCount \leftarrow 0; fCapacity \leftarrow 0; fSorted \leftarrow false;
  fDuplicate \leftarrow dupError; SetCapacity(aCapacity);
  end:
constructor MStringList. MoveStringList(var aAnother : MStringList);
  begin MObject.Init; fCount \leftarrow aAnother.fCount; fCapacity \leftarrow aAnother.fCapacity;
  fSorted \leftarrow aAnother.fSorted; fList \leftarrow aAnother.fList; fDuplicate \leftarrow aAnother.fDuplicate;
     { Empty out the other list }
  aAnother.fCount \leftarrow 0; \ aAnother.fCapacity \leftarrow 0; \ aAnother.fList \leftarrow nil;
  end;
See also sections 457 and 460.
This code is used in section 309.
       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). This hard deletes each string using
DisposeStr (§484).
destructor MStringList.Done;
  var I: integer;
  begin inherited Done;
  for I \leftarrow 0 to fCount - 1 do \langle Hard delete entry I = 456 \rangle;
  fCount \leftarrow 0; SetCapacity(0);
  end;
456. \langle Hard delete entry I 456\rangle \equiv
  with fList \uparrow [I] do { free fList \uparrow [I] }
     begin DisposeStr(fString);
     if fObject \neq nil then Dispose(fObject, Done);
     end
```

This code is used in section 455.

 $\S457$ Mizar Parser STRING LIST 119

457. 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.

```
⟨String list implementation 454⟩ +≡
function MStringList.AddString(const aStr: string): integer;
var lResult: integer;
begin ⟨Set lResult to the index of the newly inserted string 458⟩;
InsertItem(lResult, aStr); AddString ← lResult;
end;
```

458. 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 458\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 459\rangle; end

This code is used in section 457.
```

459. 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 \text{ De-duplicate a string list } 459 \rangle \equiv
\mathbf{case} \ fDuplicate \ \mathbf{of}
dupIgnore: Exit;
dupError: StringListError(coDuplicate, 0);
\mathbf{endcases}
```

This code is used in section 458.

460. 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 \text{String list implementation } 454 \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;
```

461. Merging 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;
```

120 STRING LIST Mizar Parser $\S462$

462. Clear a string list. We can hard 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;
```

463. Hard delete 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.

```
define assert\_valid\_index(\#) \equiv \mathbf{if} \ (\# < 0) \lor (\# \ge fCount) \ \mathbf{then} \ StringListError(coIndexError, \#)

procedure MStringList.Delete(aIndex:integer);

begin assert\_valid\_index(aIndex); \ DisposeStr(fList\uparrow[aIndex].fString); \ dec(fCount);

if aIndex < fCount \ \mathbf{then}

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

end;
```

464. 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 assert\_valid\_index(Index1); assert\_valid\_index(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:
```

465. Find an entry by bisection search. We can use bisection search to find the needle in the haystack. Note that this implementation seeks the *leftmost occurrence* of the needle when duplicates are allowed.

[The **if** $fDuplicate \neq dupAccept$ statement should have an **else** branch to break the loop, to avoid needless searching.]

```
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 { current item in haystack \geq aStr }
        H \leftarrow I - 1; { so look in left-half of haystack }
       if C = 0 then
          begin lResult \leftarrow True;
          if fDuplicate \neq dupAccept then L \leftarrow I;
          end;
        end;
     end:
  aIndex \leftarrow L; \ Find \leftarrow lResult;
  end;
```

 $\S466$ Mizar Parser STRING LIST 121

466. 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:
```

467. 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.

```
function MStringList.GetString(aIndex:integer): string;

begin assert\_valid\_index(aIndex); GetString \leftarrow ``;

if fList\uparrow[aIndex].fString \neq nil then GetString \leftarrow fList\uparrow[aIndex].fString\uparrow;

end;
```

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

```
function MStringList.GetObject(aIndex:integer): PObject;
begin assert\_valid\_index(aIndex); GetObject \leftarrow fList\uparrow[aIndex].fObject;
end;
```

469. 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 (§324), 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;
```

470. 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;
```

471. Value for a key. This appears to duplicate code from GetObject (§468).

```
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;
```

122 STRING LIST Mizar Parser $\S472$

472. Insert a string at an index. This seems to involve duplicate code as *AddString* (§457), 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);
assert_valid_index(aIndex); InsertItem(aIndex, aStr);
end;
```

473. 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.

There is a mirage the reader must take care about: although it superficially appears like there's bug here when the user gives aIndex > fCapacity, this private method is called only after checking the index is within bounds.

This procedure appears not to be used anywhere.

```
{REQUIRES: aIndex is within bounds}

procedure MStringList.InsertItem(aIndex: integer; const aStr: string); {private}

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;
```

474. 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;
```

475. Insert a key-value entry at a specific index.

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

476. 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;
```

§477 Mizar Parser STRING LIST 123

477. 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);
assert\_valid\_index(aIndex); fList\uparrow[aIndex].fString \leftarrow NewStr(aStr);
end;
```

478. 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 assert\_valid\_index(aIndex); fList\uparrow[aIndex].fObject \leftarrow aObject; end;
```

479. Quicksorting a string list. We have discussed quicksort to death earlier (§398), so I will omit the discussion. Also, quicksort occurs in one other place (§553).

```
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) \text{ 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);
        { Invariant: for 0 \le h < I we have CompareStr(fList \uparrow [h].fString \uparrow, P) < 0 }
        { Invariant: for J < k \le Count - 1 we have CompareStr(fList \uparrow [k].fString \uparrow, P) > 0 }
        if I \leq J then
          begin ExchangeItems(I, J); inc(I); dec(J); end;
        { Invariants: for 0 \le h \le I we have CompareStr(fList \uparrow [h].fString \uparrow, P) < 0 }
        { Invariant: for J \leq k \leq Count - 1 we have CompareStr(fList \uparrow [k].fString \uparrow, P) > 0 }
     until I > J;
     if L < J then QuickSort(L, J); {quicksort the left half}
     L \leftarrow I; { quicksort the right half }
  until I > R;
  end;
```

124 STRING LIST Mizar Parser $\S480$

480. 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:
       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:
       Sorting. This is a wrapper around the quicksort function (\S479), invoked when the fSorted flag is
482.
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;
       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;
```

484. 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;
```

 $\{485$ Mizar Parser TUPLES OF INTEGERS 125

Section 10.14. TUPLES OF INTEGERS

end;

485. The remainder of mobjects.pas focuses on finite sets of integers, finite maps of integers, and similar structures for tuples of integers. We introduce a collection of pairs of integers *specifically* to introduce the *IntRel* subclass (needed for the Equalizer, Analyzer, and Identify modules).

Also, despite the name "sequence", we should resist the intuition of "sequences from Mathematical Analysis" like $x_n = (-1)^n/n!$. Instead we should "think like programmers" and use the computer science intuition that a "sequence is a [finite] list".

```
\langle \text{Public interface for mobjects.pas } 310 \rangle + \equiv
     { Partial integers Functions }
  IntTriplet = \mathbf{record} \ X1, X2, Y: integer;
    end:
const\ MaxIntPairSize = MaxSize\ div\ SizeOf\ (IntPair);
  MaxIntTripletSize = MaxSize \ div \ SizeOf(IntTriplet);
       Now, this is the remainder of the interface
\langle \text{Public interface for mobjects.pas } 310 \rangle + \equiv
type IntPairListPtr = \uparrow IntPairList;
  IntPairList = array [0...MaxIntPairSize - 1] of IntPair;
  IntPairSeqPtr = \uparrow IntPairSeq;
  IntPairSeq = \mathbf{object} \ (MObject)
    Items: IntPairListPtr;
    Count: integer; { logical size }
    Limit: integer; { capacity }
    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;
487. First, we have a helper function for flagging errors. This should be a protected method, since it's
used internally by the IntPairList class and its subclasses but should not be used by anyone else.
\langle \text{ Tuples of integers } 487 \rangle \equiv
    { Pairs of an integers }
procedure IntPairSeq.NatSetError(Code, Info: integer);
  begin RunError(212 - Code); end;
See also sections 488, 489, 490, 491, 492, 495, and 496.
       Constructor. The empty sequence constructor, initialized to have its capacity be aLimit. Note
that SetLimit ensures aLimit is non-negative (the SetLimit method will not allow aLimit < Count and, if
this occurs, updates aLimit \leftarrow Count).
\langle Tuples of integers 487\rangle + \equiv
constructor IntPairSeq.Init(aLimit:integer);
  begin MObject.Init; Items \leftarrow nil; Count \leftarrow 0; Limit \leftarrow 0; SetLimit(aLimit);
```

126 TUPLES OF INTEGERS Mizar Parser $\S489$

489. Destructor. Hard delete the contents of the caller. Specifically, this will free the array of pointers, but not dispose the objects whose addresses are stored in the pointers.

```
\langle Tuples of integers 487\rangle +\equiv destructor IntPairSeq.Done; begin Count \leftarrow 0; SetLimit(0); end;
```

490. Insert an element. We begin by asserting the logical size is less than the *MaxIntPairSize*. We ensure the capacity of the caller's array. Then we append the *IntPair* to the collection of items, and increment the logical size of the caller's array. No sorting occurs. Duplicates are allowed.

```
\langle \text{Tuples of integers } 487 \rangle +\equiv
procedure IntPairSeq.Insert(\mathbf{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:
```

491. Soft delete an element at an index. We soft delete an element by a specific index. We assert the index is valid. Assuming so, if the index is not the last element, we just shift all entries in the underlying array to the left by one. Then we decrement the logical size of the caller.

492. Ensure capacity for an IntPair sequence. As per usual, when ensuring the capacity of a dynamic array, we monotonically increase its size. We can use this to delete all elements of the array by manually assigning the logical size $Count \leftarrow 0$ before invoking SetLimit(0).

```
⟨Tuples of integers 487⟩ +≡

procedure IntPairSeq.SetLimit(aLimit:integer);

var aItems: IntPairListPtr;

begin {ensure Count \le aLimit \le MaxIntPairSize}

if aLimit < Count then aLimit \leftarrow Count;

if aLimit > MaxIntPairSize then ALimit \leftarrow MaxIntPairSize;

if aLimit \ne Limit then

begin if ALimit = 0 then AItems \leftarrow nil

else ⟨Allocate a new array for IntPairSeq, and copy the contents over 493⟩;

⟨Free the old array, if it isn't nil 494⟩;

Items \leftarrow aItems; Limit \leftarrow aLimit;

end;

end;
```

 $\S493$ Mizar Parser TUPLES OF INTEGERS 127

```
493.
        \langle Allocate a new array for IntPairSeq, and copy the contents over 493\rangle \equiv
  begin GetMem(AItems, ALimit * SizeOf(IntPair));
  if (Count \neq 0) \land (Items \neq nil) then Move(Items \uparrow, aItems \uparrow, Count * SizeOf(IntPair));
  end
This code is used in section 492.
494. \langle Free the old array, if it isn't nil 494 \rangle \equiv
  if Limit \neq 0 then FreeMem(Items, Limit * SizeOf(IntPair))
This code is used in section 492.
       Soft delete all entries. We just set the logical size to zero. It leaves everything else untouched.
\langle Tuples of integers 487\rangle + \equiv
procedure IntPairSeq.DeleteAll;
  begin Count \leftarrow 0; end;
       Append a pair of integers. We create a new IntPair using X and Y, then append it to the caller.
\langle \text{ Tuples of integers } 487 \rangle + \equiv
procedure IntPairSeq.AssignPair(X, Y : integer);
  var lIntPair: IntPair;
  begin lIntPair.X \leftarrow X; lIntPair.Y \leftarrow Y; Insert(lIntPair);
  end;
```

This code is used in section 309.

Section 10.15. RELATIONS OF INTEGERS AS FINITE SETS

497. The basic idea is we want have a finite relation over integers $R \subseteq \mathbb{Z} \times \mathbb{Z}$ implemented in PASCAL as a sorted finite list of IntPair objects. One possible use for such a thing: we will translate identifiers into numbers (think "de Bruijn indices"), and we want to track attribute implication. This could be handled with an IntRel object.

```
This is used in the iocorrel.pas, identify.pas, the Equalizer, the Analyzer, and a polynomial library.
\langle \text{Public interface for mobjects.pas } 310 \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;
498.
       Constructor. This is just the inherited constructor (§488).
\langle \text{Int relation implementation } 498 \rangle \equiv
{ IntRel }
constructor IntRel.Init(aLimit: integer);
  begin inherited Init(aLimit);
See also sections 499, 500, 501, 502, 503, 504, and 505.
```

499. Inserting an entry. Since we are trying to describe a binary relation of integers, we want to avoid duplicate entries in the underlying array. So we need to check that *altem* is not already present in the caller's array, and then (assuming its absence) insert the item.

An important invariant: If the caller's array of *Items* was sorted before this function was called, then it will remain sorted after the function has returned control to the caller.

```
⟨Int relation implementation 498⟩ +≡
procedure IntRel.Insert(const aItem: IntPair);
var I: integer;
begin if \neg Search(aItem.X, aItem.Y, I) then
begin { Assert: index I must be within bounds }
assert\_valid\_index\_for\_nat\_set(I);
{ Note: this should be AtInsert(I, aItem) }
if Count \ge MaxIntPairSize then NatSetError(coOverflow, 0);
{ Finished with the possible errors }
if Limit = Count then SetLimit(Limit + GrowLimit(Limit));
if I \ne Count then Move(Items↑[I], Items↑[I + 1], (Count - I) * SizeOf(IntPair));
Items↑[I] \leftarrow aItem; inc(Count);
end;
end;
```

end;

Mizar Parser

500. Insert at a specific index. We can insert an entry at a specific location in the underlying array, but this breaks the contract that the underlying array is sorted using the *CompareIntPairs* function. This violates the assumption underlying the *Search* function.

```
violates the assumption underlying the Search function.
  It is used by the polynom.pas module, so we can't just ignore it.
\langle \text{Int relation implementation } 498 \rangle + \equiv
procedure IntRel.AtInsert(aIndex: integer; const aItem: IntPair);
  begin assert_valid_index_for_nat_set(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;
501. Bisection search for a relation. Search through IntRel for an entry (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.
  Assumes the underlying array is sorted using CompareIntPairs.
\langle Int relation implementation 498\rangle + \equiv
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;
       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.
\langle Int relation implementation 498\rangle + \equiv
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:
503. Index of a relation. This will return the index of the (X,Y) entry. If it is absent from the caller,
then return -1.
\langle Int relation implementation 498\rangle + \equiv
function IntRel.IndexOf(X, Y : integer): integer;
  var I: integer;
  begin IndexOf \leftarrow -1:
  if Search(X, Y, I) then IndexOf \leftarrow I;
```

504. Test for membership. This just tests if (X,Y) is contained in the caller.

```
\langle Int relation implementation 498 \rangle + \equiv function IntRel.IsMember(X,Y:integer): Boolean; var I:integer; begin IsMember \leftarrow Search(X,Y,I); end;
```

505. If (X,Y) belongs to the caller, then we're good: we do not need to do anything. Otherwise, the parent class's AssignPair (§496) method is invoked, which will just invoke the Insert method for inserting the pair to the caller.

An important invariant: if the array of *Items* is sorted before the function is called, then it remains sorted when this function is terminated.

```
\langle Int relation implementation 498\rangle +\equiv procedure IntRel.AssignPair(X,Y:integer); begin if IsMember(X,Y) then exit; inheritedAssignPair(X,Y); end;
```

 $\S506$ Mizar Parser FINITE SETS OF INTEGERS 131

Section 10.16. FINITE SETS OF INTEGERS

end;

506. We treat a finite relation of integers as a set, ignoring the second component of the entries, assigning them to zero when inserting a number. So we have $\{(x,0),\ldots,(x',0)\}$ encoded by a sorted array of integer pairs.

```
\langle \text{Public interface for mobjects.pas } 310 \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;
507. Constructor. The empty NatSet can be constructed with the usual initialization.
\langle Partial integer function implementation 507\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:
See also section 527.
This code is used in section 309.
508. 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);
```

132

Destructor. This delegates the heavy work to SetLimit(0). **destructor** NatSet.Done; **begin** $Count \leftarrow 0$; SetLimit(0); end:

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 assert\_valid\_index\_for\_nat\_set(I);
    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;
```

511. Equality of IntPair objects. This private function just tests the componentwise equality of two IntPair objects.

```
function Equals(Key1, Key2 : IntPair): Boolean;
  begin Equals \leftarrow (Key1.X = Key2.X) \land (Key1.Y = Key2.Y);
  end;
```

Search. This is a bisection search for any relation of the form (X,Y) for some Y. Assumes the array is sorted by the first component.

```
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;
```

513. 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;
  end;
```

 $\S514$ Mizar Parser FINITE SETS OF INTEGERS 133

514. 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 \leftarrow 0; fFunc.Items \leftarrow nil; end;
```

515. Union operation. We can merge another *NatSet* into the caller.

```
procedure NatSet.EnlargeBy(\mathbf{const}\ fAnother:\ NatSet);
var I:\ integer;
begin for I \leftarrow 0 to fAnother.Count - 1 do InsertElem(fAnother.Items \uparrow [i].X);
end:
```

516. 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;
```

517. Take intersection. This computes $Self \leftarrow Self \cap Other$

```
 \begin{array}{ll} \textbf{procedure} \ \textit{NatSet}. \textit{IntersectWith}(\textbf{const} \ \textit{fAnother}: \ \textit{NatSet}); \\ \textbf{var} \ \textit{k}: \ \textit{integer}; \\ \textbf{begin} \ \textit{k} \leftarrow 0; \\ \textbf{while} \ \textit{k} \leftarrow 0; \\ \textbf{while} \ \textit{k} < \textit{Count} \ \textbf{do} \\ \textbf{if} \ \neg \textit{fAnother}. \textit{HasInDom}(\textit{Items} \uparrow [k].X) \ \textbf{then} \ \textit{AtDelete}(k) \\ \textbf{else} \ \textit{inc}(k); \\ \textbf{end}; \end{array}
```

518. Insert an element. We can insert (X,0) into the caller.

```
procedure NatSet.InsertElem(X:integer); var UntPair: IntPair; begin UntPair.X \leftarrow X; UntPair.Y \leftarrow 0; Insert(UntPair); end;
```

519. 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:
```

520. We can test if an element X is in the domain of the caller.

```
function NatSet.HasInDom(fElem:integer): Boolean;
var I: integer;
begin HasInDom \leftarrow SearchPair(fElem, I);
end;
```

§521

Mizar Parser

134

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

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

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) \vee \neg Equals(fFunc.Items \uparrow [j], Items \uparrow [i]) then exit;
     end:
  IsSubsetOf \leftarrow true;
  end;
```

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;
```

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 \langle Return false if any item of <math>fFunc is in Self 525 \rangle
   else \langle \text{Return } false \text{ if any item of } Self \text{ is in } fFunc 526 \rangle;
   Misses \leftarrow true;
   end:
525. \langle \text{Return } false \text{ if any item of } fFunc \text{ is in } Self \text{ 525} \rangle \equiv
   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
This code is used in section 524.
```

 $\S526$ Mizar Parser FINITE SETS OF INTEGERS 135

```
526. \langle \text{Return } false \text{ if any item of } Self \text{ is in } fFunc | 526 \rangle \equiv  begin for k \leftarrow 0 to Count - 1 do if fFunc.SearchPair(Items \uparrow [k].X, I) then begin Misses \leftarrow false; exit \text{ end}; end This code is used in section 524.
```

527. 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.

```
\langle Partial integer function implementation 507\rangle +\equiv function NatSet.ElemNr(X:integer):integer; var I:integer; begin ElemNr \leftarrow -1; if SearchPair(X,I) then ElemNr \leftarrow I; end:
```

Mizar Parser

Section 10.17. 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 } 310 \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 } 529 \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 fFunc.Count; \ nConsistent \leftarrow 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 sections 534 and 546.
This code is used in section 309.
```

530. 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).

The implementation amounts to something like,

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

with the condition that when $y_1 = 0$, $y = y_2$ (and similarly $y_2 = 0$ implies $y = y_1$), and the symmetric difference

$$f \Delta g = (f \setminus g) \cup (g \setminus f).$$

This is not quite the same as the least common multiple in the ring N^N .

```
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;
                 \{m = \max(f(i), g(i))\}
     0: begin
       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]);
  end:
```

531. 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 (§520) which depends on SearchPair (§512) 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;
```

Mizar Parser

138

```
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 \langle \text{Insert}(X,1) | 533 \rangle;
     end;
  end:
533. \langle \text{Insert } (X,1) | 533 \rangle \equiv
  begin lIntPair.X \leftarrow X; lIntPair.Y \leftarrow 1; Insert(lIntPair);
  end
This code is used in section 532.
534. 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.
\langle NatFunc \text{ implementation } 529 \rangle + \equiv
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;
535. 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;
536. 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;
  begin inherited Done; nConsistent \leftarrow false
  end;
```

537. 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;
```

538. 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;

539. 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;
```

540. 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;
```

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

542. 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;
```

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

If there are more elements in the caller f than the other function g, $||f|| \le ||g||$, 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.

Otherwise, if there are more elements in the other function g than the caller $||g|| \le ||f||$, for each $x \in \text{dom}(g)$ if $x \notin \text{dom}(f)$, then return 0. If $f(x) \ne g(x)$, then this will 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 (§540).

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; { returns 0 }
       if Items \uparrow [k].Y \neq fNatFunc.Value(i) then exit; { returns 0 }
       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; { returns 0 }
       if fNatFunc.Items \uparrow [k].Y \neq Value(i) then exit; { returns 0 }
       end;
     Compare With \leftarrow 1; exit;
     end;
  end;
```

Mizar Parser

544. 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

$$q(x) > High(integer) - f(x)$$

```
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 \langle Has overflow occurred in NatFunc.Add? 545\rangle 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? 545\rangle \equiv
  Items \uparrow [l].Y > (High(integer) - aFunc.Items \uparrow [k].Y)
```

546. 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 } 529 \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;
```

This code is used in section 544.

Section 10.18. SEQUENCES OF NATURAL NUMBERS

```
547. A finite sequence of natural numbers is a kind of finite partial function between natural numbers.
  This is used in the first_identification.pas, inlibr.pas, and impobjs.pas.
\langle \text{Public interface for mobjects.pas } 310 \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;
548. Constructor.
\langle \text{NatSeq implementation } 548 \rangle \equiv
constructor NatSeq.InitNatSeq(ALimit, ADelta: integer);
  begin inherited Init(ALimit, ADelta); nConsistent \leftarrow true;
  end;
This code is used in section 309.
549. 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);
  end;
550. 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 } \}
     if
     (fElem < count) then Value \leftarrow Items \uparrow [fElem].Y
  else Value \leftarrow 0;
     end;
      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;
```

Section 10.19. INTEGER SEQUENCES

552. Note: this is another class describing integer sequences, and it derives directly as a subclass of MObject. It's another dynamic array.

```
\langle \text{Public interface for mobjects.pas } 310 \rangle + \equiv
  IntegerListPtr = \uparrow IntegerList;
  IntegerList = array [0..MaxIntegerListSize - 1] of integer;
  PIntSequence = \uparrow IntSequence;
  IntSequencePtr = PIntSequence;
  IntSequence = 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;
```

 $\S553$ Mizar Parser INTEGER SEQUENCES 145

553. We will need to quicksort lists of integers. This will mutate the aList argument, making it sorted. See also §398 and §479.

```
This procedure does not appear to be used anywhere in Mizar.
\langle IntSequence \text{ implementation } 553 \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);
               { Invariant: for L \leq h < I, we have CompareInt(aList \uparrow [h], P) < 0 }
       while CompareInt(aList\uparrow[J], P) > 0 do dec(J);
               { Invariant: for J < k \le R we have CompareInt(aList \uparrow [k], P) > 0 }
       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;
             { Invariant: for L < h < I, we have CompareInt(aList \uparrow [h], P) < 0 }
             { Invariant: for J \le k \le R we have CompareInt(aList \uparrow [k], P) > 0 }
     until I > J:
     if L < J then IntQuickSort(aList, L, J);
     L \leftarrow I;
  until I > R;
  end:
This code is used in section 309.
        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;
       Copy constructor. We can copy an existing sequence by simply creating an empty sequence and
inserting everything from aSeq into the newly created IntSequence object.
constructor IntSequence.CopySequence(const aSeq: IntSequence);
  begin Init(aSeq.fCapacity); AddSequence(aSeq);
  end;
```

556. 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. This just updates aSeq's logical size and capacity parameters, moves pointers around, and assigns $aSeq.fList \leftarrow nil$. Nothing is "lost", but the data is "under new management".

```
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;
```

557. Destructor. Free the inherited data, set the logical size to zero, then resize the array (to free up the array data).

```
destructor IntSequence.Done;

begin inherited.Done; fCount \leftarrow 0; SetCapacity(0);

end;
```

558. Appending an element. Given a finite sequence of integers (a_0, \ldots, a_{n-1}) , we can append a value 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;
```

559. 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(\mathbf{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:
```

560. Clearing a sequence. Update the caller's logical size to zero, then resize the underlying array to consist of zero elements.

```
procedure IntSequence.Clear;
begin if fCount \neq 0 then
begin fCount \leftarrow 0; SetCapacity(0);
end;
end;
```

561. Soft 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); dec(fCount);
```

if aIndex < fCount then $Move(fList \uparrow [aIndex + 1], fList \uparrow [aIndex], (fCount - aIndex) * SizeOf(integer));$ end;

562. We report errors using this helper function.

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

563. 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;
```

 $\S564$ Mizar Parser INTEGER SEQUENCES 147

564. For a finite sequence (a_0, \ldots, a_{n-1}) and a value x, if there is some entry $a_i = x$ with $a_j \neq x$ for all j < i, then return i. Otherwise return -1. That is to say, returns the first index of an entry matching the given value (if any), defaults to -1 when there is no entry.

```
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;
565. 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;
566. 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;
  end;
      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));
     if fCapacity \neq 0 then FreeMem(fList, fCapacity * SizeOf(integer));
     fList \leftarrow lList; fCapacity \leftarrow aCapacity;
     end:
  end;
```

148 INTEGER SETS Mizar Parser §568

Section 10.20. INTEGER SETS

568. A finite set of integers is represented by a finite sequence of integers which has no duplicate entries. In other words, an injective finite sequence of integers.

```
 \begin{array}{l} \langle \, {\rm Public \; interface \; for \; mobjects.pas \; 310} \, \rangle \, + \equiv \\ PIntSet = \, \uparrow IntSet; \\ IntSetPtr = pIntSet; \\ IntSet = {\rm object \; } (IntSequence) \\ {\rm \; function \; } Insert(aInt:integer): \; integer; \; virtual; \\ {\rm \; function \; } DeleteInt(aInt:integer): \; integer; \; virtual; \\ {\rm \; function \; } Find(aInt:integer): \; var \; aIndex:integer): \; Boolean; \; virtual; \\ {\rm \; function \; } IndexOf(aInt:integer): \; integer; \; virtual; \\ {\rm \; procedure \; } AtInsert(aIndex, aInt:integer); \; virtual; \\ {\rm \; function \; } IsInSet(aInt:integer): \; Boolean; \; virtual; \\ {\rm \; function \; } IsEqualTo({\rm const \; } aSet: \; IntSet): \; Boolean; \; virtual; \\ {\rm \; function \; } IsSubsetOf({\rm const \; } aSet:IntSet): \; Boolean; \; virtual; \\ {\rm \; function \; } IsSupersetOf({\rm var \; } aSet:IntSet): \; Boolean; \; virtual; \\ {\rm \; function \; } Misses({\rm var \; } aSet:IntSet): \; Boolean; \; virtual; \\ {\rm \; function \; } Misses({\rm var \; } aSet:IntSet): \; Boolean; \; virtual; \\ {\rm \; end \; } ; \\ \end \; ; \end{array}
```

569. Insert an element. 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.

Ensures the resulting array is sorted, provided it was sorted before this function was called.

```
\langle \mathit{IntSet} \; \mathsf{Implementation} \; 569 \rangle \equiv
\mathsf{function} \; \mathit{IntSet} \; \mathit{Insert} (\mathit{aInt} : \mathit{integer}) \colon \mathit{integer};
\mathsf{var} \; \mathit{Undex} \colon \mathit{integer};
\mathsf{begin} \; \mathsf{if} \; \mathit{Find} (\mathit{aInt}, \mathit{Undex}) \; \mathsf{then} \quad \{ \; \mathsf{already} \; \mathsf{contains} \; \mathsf{the} \; \mathsf{element?} \}
\mathsf{begin} \; \mathit{Insert} \; \leftarrow \; \mathit{UIndex}; \; \mathit{exit} \; \mathsf{end};
\mathsf{if} \; \mathit{fCount} = \mathit{fCapacity} \; \mathsf{then} \; \; \mathit{SetCapacity} (\mathit{fCapacity} + \mathit{GrowLimit} (\mathit{fCapacity}));
\mathsf{if} \; \mathit{UIndex} \; < \; \mathit{fCount} \; \; \mathsf{then} \; \; \mathit{Move} (\mathit{fList} \uparrow [\mathit{UIndex}], \mathit{fList} \uparrow [\mathit{UIndex} + 1], (\mathit{fCount} - \mathit{UIndex}) * \mathit{SizeOf} (\mathit{integer}));
\mathit{fList} \uparrow [\mathit{UIndex}] \; \leftarrow \; \mathit{aInt}; \; \mathit{inc} (\mathit{fCount}); \; \mathit{Insert} \; \leftarrow \; \mathit{UIndex};
\mathsf{end};
This code is used in section 309.
```

570. Soft delete an element. 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;
```

 $\S571$ Mizar Parser INTEGER SETS 149

571. Finding an element. 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:
       Index of element. 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;
      The AtInsert method is "grandfathered in", but not supported, so we raise an error if anyone tries
using it.
procedure IntSet.AtInsert(aIndex, aInt: integer);
  begin IntListError(coSortedListError, 0);
  end;
       Test for membership. 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:
575. Set equality. 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. (This is established by
Theorem CARD_2:102.)
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;

150 INTEGER SETS Mizar Parser $\S576$

576. Subset predicate. We can test $A \subseteq B$ by $|A| \le |B|$ and for each $a \in A$ we have $a \in B$. (This is a consequence of Theorem CARD_2:48.) 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;
```

577. Superset predicate. We have $A \supset B$ if $B \subseteq A$.

```
function IntSet.IsSupersetOf(\mathbf{var}\ aSet:IntSet): Boolean; \mathbf{begin}\ IsSupersetOf \leftarrow aSet.IsSubsetOf(Self); \mathbf{end};
```

578. Test for disjointness. We have $A \cap B = \emptyset$ if every $a \in A$ is such that $a \notin B$. There are fewer iterations in the loop if we test every element in the smaller set is not a member of the larger set. This trick works because "misses" is a symmetric predicate.

```
function IntSet.Misses(\mathbf{var}\ aSet:IntSet): Boolean;
\mathbf{var}\ k:\ integer;
\mathbf{begin}\ if\ fCount > aSet.fCount\ then
\mathbf{begin}\ for\ k \leftarrow 0\ to\ aSet.fCount - 1\ do
\mathbf{if}\ IsInSet(aSet.fList\uparrow[k])\ then
\mathbf{begin}\ Misses \leftarrow false;\ exit\ end
\mathbf{end}
\mathbf{else}\ \mathbf{begin}\ for\ k \leftarrow 0\ to\ fCount - 1\ do
\mathbf{if}\ aSet.IsInSet(fList\uparrow[k])\ then
\mathbf{begin}\ Misses \leftarrow false;\ exit\ end;
\mathbf{end};
Misses \leftarrow true;
\mathbf{end};
```

end;

Section 10.21. PARTIAL BINARY INTEGER FUNCTIONS

```
579. 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.
  We stress the class hierarchy: the parent class is MObject itself.
\langle \text{Public interface for mobjects.pas } 310 \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;
      We have a convenience function for reporting errors.
\langle \text{ Partial Binary integer Functions } 580 \rangle \equiv
procedure BinIntFunc.BinIntFuncError(aCode, aInfo: integer);
  begin RunError(212 - aCode); end;
This code is used in section 309.
581. Constructor. We initialize the empty partial function, and create an array whose capacity is
initialized to aLimit.
constructor BinIntFunc.Init(aLimit:integer);
  begin MObject.Init; fList \leftarrow \mathbf{nil}; fCount \leftarrow 0; fCapacity \leftarrow 0; SetCapacity(aLimit);
  end;
582. Destructor. We assign the logical size of the caller to zero, then resize the underlying array to zero
elements (i.e., free everything).
destructor BinIntFunc.Done;
  begin fCount \leftarrow 0; SetCapacity(0);
```

583. Insert an entry. If we have a partial function $f: \mathbf{Z} \times \mathbf{Z} \to \mathbf{Z}$ and a triple (x_1, x_2, y) , then check if $(x_1, x_2) \in \text{dom}(f)$. If so, we're done — this will not overwrite the value at that argument.

Otherwise we add $f(x_1, x_2) = y$ to the partial function.

```
procedure BinIntFunc.Insert(\mathbf{const}\ aItem:\ IntTriplet);
var I:\ integer;
begin if \neg Search(aItem.X1, aItem.X2, I) then \{(x_1, x_2) \notin \mathrm{dom}(f)\}
begin if (I < 0) \lor (I > fCount) then \{\mathrm{index}\ \mathrm{out}\ \mathrm{of}\ \mathrm{bounds}\}
begin BinIntFuncError(coIndexError, 0);\ exit;\ \mathbf{end};
if fCapacity = fCount\ \mathbf{then}\ SetCapacity(fCapacity + GrowLimit(fCapacity));
if I \neq fCount\ \mathbf{then}\ Move(fList\uparrow[I], fList\uparrow[I+1], (fCount-I) * SizeOf(IntTriplet));
fList\uparrow[I] \leftarrow aItem;\ inc(fCount);
end;
end;
```

584. Delete an entry. 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 \ge 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;
```

585. Ensure capacity. We need to ensure $fCount \leq aLimit \leq MaxIntTripletSize$, by assigning aLimit to be fCount (when aLimit < fCount) or MaxIntTripletSize (when aLimit is bigger).

We allocate a new array (when the new capacity is nonzero) and copy the data over to the new array. Then we delete the old array if it exists (tested by $fCapacity \neq 0$). Then we just update the caller's fList pointer and capacity field.

```
procedure BinIntFunc.SetCapacity(aLimit:integer);
var aItems: IntTripletListPtr;
begin { Ensure fCount \le aLimit \le MaxIntTripletSize }

if aLimit < fCount then aLimit \leftarrow fCount;
if aLimit > MaxIntTripletSize then ALimit \leftarrow MaxIntTripletSize;
if aLimit \ne fCapacity then { allocate a new array, copy data over }

begin if ALimit = 0 then AItems \leftarrow nil
else begin GetMem(AItems, ALimit * SizeOf(IntTriplet));
if (fCount \ne 0) \land (fList \ne nil) then Move(fList \uparrow, aItems \uparrow, fCount * SizeOf(IntTriplet));
end;
if fCapacity \ne 0 then FreeMem(fList, fCapacity * SizeOf(IntTriplet));
fList \leftarrow aItems; fCapacity \leftarrow aLimit;
end;
end;
```

586. Hard delete all entries. 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;
```

```
Search for entry. 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. Remember Move is misnamed
because of Borland, it really copies data (despite what the name would lead you to believe).
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;
       Index of entry. Given f: \mathbb{Z} \times \mathbb{Z} \to \mathbb{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;
  end;
        Test if defined on pair. Test if (x_1, x_2) \in dom(f).
function BinIntFunc.HasInDom(X1, X2 : integer): Boolean;
  var I: integer;
  begin HasInDom \leftarrow Search(X1, X2, I);
  end;
591. Insert an entry. 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
     end:
  lIntTriplet.X1 \leftarrow X1; lIntTriplet.X2 \leftarrow X2; lIntTriplet.Y \leftarrow Y; Insert(lIntTriplet);
```

Mizar Parser

154

```
Increment value at argument. 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. 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;
        Decrement value at argument. 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);
        Return value for argument. 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;
595. Add two partial functions together. Given two partial functions f, g: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}, compute
f + q: \mathbf{Z} \times \mathbf{Z} \longrightarrow \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;
```

596. Sum all values. For $f: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, we compute

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

```
\begin{array}{l} \textbf{function} \ BinIntFunc.CountAll: integer; \\ \textbf{var} \ k,l: \ integer; \\ \textbf{begin} \ l \leftarrow 0; \\ \textbf{for} \ k \leftarrow 0 \ \textbf{to} \ fCount - 1 \ \textbf{do} \ inc(l,fList \uparrow [k].Y); \\ CountAll \leftarrow l; \\ \textbf{end;} \end{array}
```

Mizar Parser

156

Section 10.22. PARTIAL INTEGERS TO PAIR OF INTEGERS FUNCTIONS

```
597. Partial functions of the form f: \mathbf{Z} \to \mathbf{Z} \times \mathbf{Z}. Note the class hierarchy: this is a direct subclass of
MObject. This is used only in the first_identification.pas file.
\langle \text{Public interface for mobjects.pas } 310 \rangle + \equiv
  Int2PairOfInt = \mathbf{record} \ X, Y1, Y2: integer;
    end:
  Int2PairOfIntFuncPtr = \uparrow Int2PairOfIntFunc;
  Int2PairOfIntFunc = \mathbf{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;
598. We have a helper function for raising errors.
\langle Partial integers to Pair of integers Functions 598 \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 309.
599. 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 nil; fCount \leftarrow 0; fCapacity \leftarrow 0; SetCapacity(aLimit);
  end;
600. Destructor. We set the logical size to zero, then ensure the capacity of the underlying array is zero
(thereby hard deleting the array).
destructor Int2PairOfIntFunc.Done;
  begin fCount \leftarrow 0; SetCapacity(0);
  end;
```

end:

Mizar Parser

```
601. Insert an entry. Inserting (x, y_1, y_2) into f: \mathbb{Z} \to \mathbb{Z} \times \mathbb{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(\textbf{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;\ \textbf{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;\ \textbf{end};
```

602. Hard delete an entry. 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 \ge fCount) then { index out of bounds }

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;
```

603. Ensure capacity. There is no test that *aLimit* is less than some maximum size, unlike every other similar "ensure capacity" function in this file. And we use the built-in *SetLength* primitive function from PASCAL to resize the dynamic array.

```
procedure Int2PairOfIntFunc.SetCapacity(aLimit:integer);

begin if aLimit < fCount then aLimit \leftarrow fCount;

setlength(fList, aLimit); fCapacity \leftarrow aLimit;

end;
```

604. Soft delete all entries. We can "soft delete" all entries in the partial function.

```
procedure Int2PairOfIntFunc.DeleteAll; begin fCount \leftarrow 0; end;
```

Mizar Parser

158

Search for entry. We can bisection search on the domain. Observe: once the needle has been found in the haystack, we have $H \leftarrow I - 1$ and $L \leftarrow I$, which means the loop will terminate.

```
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:
606. Copy constructor. This leaves the argument aFunc unchanged. The argument should be a const
not a var ?
constructor Int2PairOfIntFunc.CopyInt2PairOfIntFunc(var aFunc: Int2PairOfIntFunc);
  begin Init(aFunc.fCapacity); Move(aFunc.fList[0], fList[0], aFunc.fCapacity * SizeOf(Int2PairOfInt));
  fCount \leftarrow aFunc.fCount;
  end;
607. Index of argument. 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;
       Test if defined at point. Test if x \in dom(f).
function Int2PairOfIntFunc.HasInDom(X:integer): Boolean;
  var I: integer;
  begin HasInDom \leftarrow Search(X, I);
  end;
      Assign an entry. Attempt to insert (x, y_1, y_2) into f: \mathbf{Z} \to \mathbf{Z} \times \mathbf{Z}. This will not overwrite a pre-
existing entry, as per the contract of Int2PairOfIntFunc.Insert (§601).
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:
```

610. Determine value at argument. Given $f: \mathbf{Z} \rightharpoonup \mathbf{Z} \times \mathbf{Z}$ 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; end else Int2PairOfIntFuncError(coDuplicate, 0); end;
```

611. Lingering random declarations in the interface. We have a myriad of random declarations, so we just stick them all here.

```
 \langle \text{Public interface for mobjects.pas } 310 \rangle + \equiv \\ \{ \text{Comparing Strings wrt MStrObj} \} \\ \text{function } \textit{CompareStringPtr}(a\textit{Key1}, a\textit{Key2} : \textit{Pointer}) \colon \textit{integer}; \\ \{ \text{Comparing Strings and integers} \} \\ \text{function } \textit{CompareStr}(a\textit{Str1}, a\textit{Str2} : \textit{string}) \colon \textit{integer}; \\ \text{function } \textit{CompareIntPairs}(X1, Y1, X2, Y2 : \textit{Longint}) \colon \textit{integer}; \\ \{ \text{Dynamic String handling routines} \} \\ \text{function } \textit{NewStr}(\textbf{const } S \colon \textit{string}) \colon \textit{PString}; \\ \text{procedure } \textit{DisposeStr}(P : \textit{PString}); \\ \text{function } \textit{GrowLimit}(a\textit{Limit} : \textit{integer}) \colon \textit{integer}; \\ \text{function } \textit{CompareNatFunc}(a\textit{Key1}, a\textit{Key2} : \textit{Pointer}) \colon \textit{integer}; \\ \text{procedure } \textit{Abstract1}; \\ \text{var } \textit{EmptyNatFunc} \colon \textit{NatFunc}; \\ \end{cases}
```

160 XML DICTIONARY Mizar Parser §612

File 11

XML Dictionary

```
612. We have several types declared in the xml_dict.pas file. These are enumerated types, and string constants for their names.  \langle \, \text{xml_dict.pas 612} \, \rangle \equiv \\ \langle \, \text{GNU License 4} \, \rangle
```

```
\( \langle \text{GNU License 4} \)
unit \( xml_\text{dict}; \)
unit \( xml_\text{dict}; \)
interface
uses \( mobjects; \)
\( \{ \text{known (and only allowed) XML elements } \}
\( \text{type } \langle \text{Types of xml_\text{dict.pas 613}} \rangle \)
\( \text{const} \langle \text{Constants of xml_\text{dict.pas 614}} \rangle \)
implementation
end.
```

613. $\langle \text{Types of xml_dict.pas } 613 \rangle \equiv$

XMLElemKind = (elUnknown, elAdjective, elAdjective Cluster, elArticleID, elAncestors, elArguments, elBlock, elConditions, elCorrectness Conditions, elDefiniens, elDirective, elEnviron, elEquality, elFieldSegment, elFormat, elFormats, elIdent, elItem, elIterativeStep, elLabel, elLink, elLoci, elLociEquality, elLocus, elNegatedAdjective, elPartialDefiniens, elPriority, elProposition, elProvisionalFormulas, elRedefine, elRightCircumflexSymbol, elSchematic Variables, elScheme, elSelector, elSetMember, elSkippedProof, elSymbol, elSymbolCount, elSymbols, elSubstitution, elTypeSpecification, elTypeList, elVariable, elVariables, elVocabularies, elVocabulary);

See also section 615.

This code is used in section 612.

See also section 616.

This code is used in section 612.

 $\S615$ Mizar Parser XML DICTIONARY 161

615. Note that atX1 and atX2 are not used anywhere in Mizar, but atY1 and atY2 are used in the first_identification.pas.

File 12

Environment library

617. 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 (§643) and CheckCompatibility (§639).

Here the assumption that there's an environment variable "\$MIZFILES" set to the directory containing the Mizar Mathematical library. We discussed environment variables earlier (§65).

```
\langle base/librenv.pas 617 \rangle \equiv
  (GNU License 4)
unit librenv;
  interface
  uses mobjects;
  const MML = 'mml'; EnvMizFiles = 'MIZFILES';
  ⟨ Public function declarations for librenv.pas 627⟩
  type (Declare FileDescr data type 619)
    \langle \text{Declare } FileDescrCollection \text{ data type } 622 \rangle
  var (Global variables declared in librenv.pas 618)
  implementation
  uses
    if_def (WIN32) windows, end_if
    mizenv\,, pcmizver\,, mconsole\,;
    ⟨Implementation for librenv.pas 620⟩
  begin InitLibrEnv; CheckCompatibility;
  end.
618. \langle Global variables declared in librenv.pas 618\rangle \equiv
MizPath: string; { path to where Mizar binaries are located }
MizFiles: string; { the "$MIZFILES" environment variable }
LocFilesCollection: FileDescrCollection;
This code is used in section 617.
```

 $\S619$ Mizar Parser FILE DESCRIPTORS 163

Section 12.1. FILE DESCRIPTORS

619. We use file descriptors for things. These are just "a file name" and "a timestamp". The "file name" is either an absolute path to a file in the Mizar mathematical library, or a relative path to a file in the user's local "./prel/" subdirectory.

Historically, the notion of "file descriptors" pre-dates UNIX. For example, R.F. Clippinger's "FACT—A Business Compiler: Description and Comparison with COBOL and Commercial Translator" (*Int. Tracts Comput. Sci. Technol. Their Appl.* 2 (1961) 231–292, doi:10.1016/B978-1-4831-9779-1.50014-8) seems to be among the earliest use of the term.

CAUTION: PASCAL's *LongInt* is a 32-bit signed integer, which means that Mizar will experience the 2038 "Y2k" bug on 03:14:07 UTC on 19 January 2038. Switching to *Int6*4 would solve this problem (well, delay it for a time longer than the lifetime of the universe).

```
⟨ Declare FileDescr data type 619⟩ ≡
PFileDescr = ↑FileDescr;
FileDescr = object (MObject)
    nName: PString;
    Time: LongInt;
    constructor Init(fIdent : string; fTime : LongInt);
    destructor Done; virtual;
end;
This code is used in section 617.
```

620. Constructor. When creating a new *FileDescr* object, we should allocate enough space for a copy of the file name. The object then "owns" this copy of the file name string, and must free it when the object is freed from memory.

```
\langle Implementation for librenv.pas 620\rangle \equiv constructor FileDescr.Init(fIdent: string; fTime: LongInt); begin nName \leftarrow NewStr(fIdent); Time \leftarrow fTime; end; See also sections 623, 628, 630, 632, 634, 638, 639, and 643. This code is used in section 617.
```

621. Destructor. We need to free the string "owned" by the file descriptor object.

```
destructor FileDescr.Done;
  begin DisposeStr(nName);
  end;
```

Section 12.2. COLLECTION OF FILE DESCRIPTIONS

```
This is just MSortedCollection (§428) of FileDescr objects.
\langle \text{ Declare } FileDescrCollection \text{ data type } 622 \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 617.
       Ordering file descriptors. Comparing two entries in a file descriptor collection amounts to com-
paring the names for the file descriptors. [This should be the CompareStringPtr (\S424) function, just to
keep the code DRY.
\langle Implementation for librenv.pas 620\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;
624. Populate time field. Inserting file's "last modified" timestamp into the file descriptors relies upon
mizenv.pas's GetFileTime (§51) 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;
625. 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, ..., 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;
```

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;
```

627. The library path tries to use the local version of a file, if it exists as tested with MFileExists (§47). 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.

```
\langle \text{Public function declarations for librenv.pas } 627 \rangle \equiv
function LibraryPath(fName, fExt : string): string;
See also sections 629, 631, 633, and 637.
This code is used in section 617.
      \langle \text{Implementation for librenv.pas } 620 \rangle + \equiv
function LibraryPath(fName, fExt : string): string;
  begin LibraryPath \leftarrow ``;
  if MFileExists('prel' + DirSeparator + fName + fExt) then { populate with local file}
    \mathbf{begin}\ LocFilesCollection.Insert(New(PFileDescr,Init(`\mathtt{prel'} + DirSeparator + fName + fExt, 0)));
    LibraryPath \leftarrow \text{`prel'} + DirSeparator + fName + fExt; exit
  if MFileExists(MizFiles + `prel' + DirSeparator + fName[1] + DirSeparator + fName + fExt) then
    LibraryPath \leftarrow MizFiles + \text{`prel'} + DirSeparator + fName[1] + DirSeparator + fName + fExt;
  end;
       This function actually is not used anywhere, so I am not sure why we have it.
\langle \text{Public function declarations for librerv.pas } 627 \rangle + \equiv
procedure ReadSortedNames (fName: string; var fList: MStringCollection); { UNUSED! }
       \langle Implementation for librenv.pas 620\rangle + \equiv
procedure ReadSortedNames (fName: string; var fList: MStringCollection); { UNUSED! }
  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;
```

This code is used in sections 634 and 638.

166

```
Again, this function is not used anywhere, so I am not sure why we have it.
⟨Public function declarations for librenv.pas 627⟩ +≡
procedure ReadNames(fName : string; var fList : StringColl); { UNUSED! }
632. (Implementation for librenv.pas 620) +\equiv
procedure ReadNames(fName: string; var fList: StringColl); { UNUSED!}
  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));
      end;
    exit;
    end;
  fList.Init(2,10); fList.Insert(NewStr(fName));
  end;
     This function is used in usrtools/lisvoc.dpr. The fList is the VocList which consists of all the
entries in "$MIZFILES/mml.vct". The fParam is either 1 or 2.
  If the user writes in a file "foobar" one vocabulary on a line (e.g., "GROUP_1" on one line, "CAT_1"
on another), then fName \leftarrow \texttt{`Qfoobar'} will lookup the contents "foobar" and load them into the fList
parameter.
\langle \text{Public function declarations for librerv.pas } 627 \rangle + \equiv
procedure GetSortedNames(fParam: byte; var fList: MStringCollection);
634. (Implementation for librenv.pas 620) +\equiv
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] = 0 then \langle Populate fList with the contents of <math>FileName and exit 635\rangle;
  \langle Populate fList with the command-line arguments 636\rangle;
  end;
      (Populate fList with the contents of FileName and exit 635) \equiv
  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
```

```
636.
       \langle \text{ Populate } fList \text{ with the command-line arguments } 636 \rangle \equiv
  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
This code is used in sections 634 and 638.
637. Continuing with the "this is not used anywhere" theme, this function is not used anywhere. The
GetNames procedure turns out to be the same as GetSortedNames.
⟨Public function declarations for librenv.pas 627⟩ +≡
procedure GetNames(fParam: byte; var fList: StringColl); { DUPLICATE CODE }
638. (Implementation for librenv.pas 620) +\equiv
\mathbf{procedure} \ \ GetNames(fParam:byte; \mathbf{var} \ fList:StringColl); \quad \{ \ \mathrm{DUPLICATE} \ \ \mathrm{CODE} \ \}
  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] = \mathfrak{C} then \langle Populate fList with the contents of <math>FileName and exit 635\rangle;
  \langle \text{ Populate } fList \text{ with the command-line arguments } 636 \rangle;
  end;
```

Mizar Parser

168

Section 12.3. 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

We should probably move towards TOML rather than INI as the format for storing information, since INI has no standard or specification but TOML has one.

```
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 } 620 \rangle + \equiv
procedure CheckCompatibility;
  var lFile: text; lLine, lVer1, lVer2, l: string; lPos, lCode: integer;
    lMizarReleaseNbr, lMizarVersionNbr, lMizarVariantNbr: integer;
  begin (Open mml.ini file 640)
  lMizarReleaseNbr \leftarrow -1; \ lMizarVersionNbr \leftarrow -1; \ lMizarVariantNbr \leftarrow -1;
  ⟨Try to read the Mizar version from mml.ini 641⟩;
  (Assert MML version is compatible with Mizar version 642)
  end;
640. We open the $MIZFILES/mml.ini file for reading.
\langle \text{ Open mml.ini file } 640 \rangle \equiv
  FileExam(MizFiles + MML + `.ini`); Assign(lFile, MizFiles + MML + `.ini`); Reset(lFile);
This code is used in section 639.
641. \langle Try to read the Mizar version from mml.ini 641\rangle
  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
```

This code is used in section 639.

642. 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.

643. 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 if \#[length(\#)] \neq DirSeparator then \# \leftarrow \# + DirSeparator; 
 \langle Implementation for librenv.pas 620 \rangle + \equiv  
 procedure InitLibrEnv; 
 begin LocFilesCollection.Init(0,20); MizPath \leftarrow ExtractFileDir(ParamStr(0)); 
 \langle Initialize MizFiles 644 \rangle 
 MizFileName \leftarrow ``; EnvFileName \leftarrow ``; ArticleName \leftarrow ``; ArticleExt \leftarrow ``; end;
```

644. 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.

```
⟨ Initialize MizFiles 644⟩ ≡
⟨ Guess MizFiles from environment variables or executable path 645⟩
if MizFiles ≠ `` then append_dir_separator(MizFiles);
if MizFiles = `` then MizFiles ← DirSeparator;
This code is used in section 643.
```

645. 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 Guess MizFiles from environment variables or executable path 645 \rangle \equiv MizFiles \leftarrow GetEnvStr(EnvMizFiles);
if MizFiles =  then MizFiles \leftarrow MizPath;
This code is used in section 644.
```

170 XML PARSER Mizar Parser $\S646$

File 13

XML Parser

646. 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"?>"

```
\langle base/xml_parser.pas 646 \rangle \equiv
  (GNU License 4)
unit xml_parser;
  interface uses mobjects, errhan;
  (Constants for xml_parser.pas 647)
  ⟨Type declarations for xml_parser.pas 648⟩
  procedure XMLASSERT(aCond : Boolean);
  procedure UnexpectedXMLElem(const aElem: string; aErr: integer);
  implementation
    mdebug uses info end_mdebug;
  (Implementation of XML Parser 650)
  end .
647.
      Constant parameters. We have a few constant parameters for the error codes.
\langle \text{Constants for xml_parser.pas } 647 \rangle \equiv
const InOutFileBuffSize = $4000;
{ for xml attribute tables }
const errElRedundant = 7500; { End of element expected, but child element found }
const errElMissing = 7501; { Child element expected, but end of element found }
const errMissingXMLAttribute = 7502; { Required XML attribute not found }
const errWrongXMLElement = 7503; { Different XML element expected }
```

const errBadXMLToken = 7506; {Unexpected XML token}

This code is used in section 646.

 $\S648$ Mizar Parser XML PARSER 171

648. 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 } 648 \rangle \equiv
type XMLTokenKind = (Err, {an error symbol})
  BI, \{ <? \}
  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 653)
  TElementState = (eStart, eEnd);  { high-level parser states, see procedure NextElementState }
  (Declare XML Attribute Object 649)
  (Declare XML Parser object 661)
This code is used in section 646.
      XML Attribute Object. An XML attribute contains the attribute name and its value. We can
represent it as "just" an MStrObj (§317) with an additional "value" field.
\langle \text{ Declare XML Attribute Object 649} \rangle \equiv
  XMLAttrPtr = \uparrow XMLAttrObj;
  XMLAttrObj = \mathbf{object} \ (MStrObj)
    nValue: string;
    constructor Init(const aName, aValue: string);
This code is used in section 648.
       Constructor. This uses the MStrObj. Init constructor to initialize the name, then it sets the value.
\langle \text{Implementation of XML Parser 650} \rangle \equiv
constructor XMLAttrObj.Init(const aName, aValue: string);
  begin inherited Init(aName); nValue \leftarrow aValue;
See also sections 651, 652, 654, 656, 657, 660, 662, 668, 670, and 671.
This code is used in section 646.
       Assertion. We have a helper function for asserting things about XML. This is just a wrapper around
MizAssert (§147).
\langle \text{Implementation of XML Parser } 650 \rangle + \equiv
procedure XMLASSERT (aCond : Boolean);
  begin MizAssert(errWrongXMLElement, aCond);
  end;
```

172 XML PARSER Mizar Parser $\S 652$

```
652. Unexpected XML Element. Another helper function for checking XML parsing.

⟨Implementation of XML Parser 650⟩ +≡
procedure UnexpectedXMLElem(const aElem: string; aErr: integer);
mdebug
var lEl: string;
end_mdebug
begin
mdebug InfoNewLine; end_mdebug
RunTimeError(aErr);
end;
```

 $\S653$ Mizar Parser XML SCANNER OBJECT 173

Section 13.1. XML SCANNER OBJECT

653. 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 653\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 648.
       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 Implementation of XML Parser 650\rangle + \equiv
constructor XMLScannObj.InitScanning(const aFileName: string);
  begin inherited Init; (Prepare to read in the contents of XML file 655);
  nSpelling \leftarrow \ \ \ \ \ nLine \leftarrow \ \ \ \ \ nCurCol \leftarrow 0; \ nPos.Line \leftarrow 0; \ nPos.Col \leftarrow 0;
  GetToken;
  end;
       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 655\rangle \equiv
  Assign(nSourceFile, aFileName); GetMem(nSourceFileBuff, InOutFileBuffSize);
  SetTextBuf(nSourceFile, nSourceFileBuff\uparrow, InOutFileBuffSize); Reset(nSourceFile) { open for reading }
This code is used in section 654.
656. Destructor. We need to close the XML file, as well as free up the input buffer.
\langle Implementation of XML Parser 650\rangle + \equiv
destructor XMLScannObj.Done;
  begin close (nSourceFile); FreeMem(nSourceFileBuff, InOutFileBuffSize);
  nLine \leftarrow ``; nSpelling \leftarrow ``;
  inherited Done;
  end;
```

657. Getting the token. The scanner produces tokens on demand. They are assembled into a tree data 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.

If the reader wants to extend Mizar to support UTF-8, then the byte parameter needs to be modified accordingly. A crude first approximation would be to set the entries indexed [128...255] to 1.

```
define update\_lexeme \equiv nSpelling \leftarrow Copy(nLine, nPos.Col, nCurCol - nPos.Col)
\langle Implementation of XML Parser 650\rangle + \equiv
procedure XMLScannObj.GetToken;
 const CharKind: array [chr(0) ... chr(255)] of
    - . / 0 1 2 3 4 5 6 7 8 9 : ; }
   0, 0, 0, 3, 0, 0, 3, 0, 0, 0, 0, 0, 0, 3, 3, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 0, 0, 0, 0, 0
   { 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 658);
 nPos.Col \leftarrow nCurCol;
 (Get token kind based off of leading character 659);
 update_lexeme:
 while (nCurCol < length(nLine)) \land (nLine[nCurCol] \in [`\_', `\_']) do inc(nCurCol);
 end:
```

658. 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 658} \rangle \equiv \\ \text{while } nCurCol = length(nLine) \text{ do} \\ \text{begin if } eof(nSourceFile) \text{ then} \\ \text{begin } nCurTokenKind \leftarrow EOTX; \ nSpelling \leftarrow ``; \ exit \text{ end}; \\ ReadLn(nSourceFile, nLine); \ inc(nPos.Line); \ nLine \leftarrow nLine + `\_`; \ nCurCol \leftarrow 1; \\ \text{while } (nCurCol < length(nLine)) \wedge (nLine[nCurCol] = `\_`) \text{ do } inc(nCurCol); \\ \text{end}
```

This code is used in section 657.

§659 Mizar Parser XML SCANNER OBJECT 175

659. 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;
            end
  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;
          ": \mathbf{begin} \ nCurTokenKind \leftarrow DT; \ inc(nCurCol); \mathbf{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 659\rangle \equiv
  case nLine[nCurCol] of
  `a`..`z`, `A`..`Z`, `O`...`9`, `_`, `-`, `&`: keep_eating_alphadigits;
  ": begin nCurTokenKind \leftarrow QT; inc(nCurCol) end:
  '=': begin nCurTokenKind \leftarrow EQ; inc(nCurCol) end;
  '<': begin get_tag_kind; end;
  \rightarrow: begin nCurTokenKind \leftarrow GT; inc(nCurCol) end;
  ': keep\_getting\_until\_end\_of\_tag(EE);
  : keep\_getting\_until\_end\_of\_tag(EI);
  othercases begin nCurTokenKind \leftarrow Err; inc(nCurCol) end;
  endcases
```

This code is used in section 657.

660. Get attribute value. Scanners can obtain attribute values as tokens. This is used by the XML parser (§§665, 668). 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 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 650 \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;
```

176 XML PARSER Mizar Parser $\S 661$

Section 13.2. XML PARSER

```
661. We recall (§648) the type for element states (it's an enumerated type with two values, eStart and
eEnd).
\langle Declare XML Parser object 661\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 NextTaq; virtual;
    procedure NextElementState; virtual;
    procedure AcceptEndState; virtual;
    procedure AcceptStartState; virtual;
    procedure OpenStartTag; virtual;
    procedure CloseStartTag; virtual;
    procedure CloseEmptyElementTag; virtual;
    procedure ProcessEndTag; virtual;
    procedure ProcessAttributeName; virtual;
    procedure ProcessAttributeValue; virtual;
    procedure SetAttributeValue(const aVal: string);
  end:
This code is used in section 648.
662. 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
           while (nCurTokenKind \neq EOTX) \land (nCurTokenKind \neq EI) do GetToken;
        if nCurTokenKind = EI then GetToken
  define skip\_all\_other\_ids \equiv
           while nCurTokenKind = BI do
             begin GetToken; skip_xml_prolog;
             end
\langle Implementation of XML Parser 650\rangle + \equiv
constructor XMLParserObj.InitParsing(const aFileName: string);
  begin inherited InitScanning(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:
```

 $\S663$ Mizar Parser XML PARSER 177

663. 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;
```

664. 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 if nCurTokenKind \neq # 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;
```

665. The parser will the consume the next tag or element. It's useful to recall the token kinds (§648). Curiously, the attributes are skipped during this parsing function.

This will be using the inherited procedure GetToken (§657).

```
{ Parses next part of XML, used for skipping some part of XML } { setting the nState to eStart or eEnd. } { nElName is set properly } { nAttrVals 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]); OpenStartTag; GetToken; end;
EE: begin nState \leftarrow eEnd; GetToken; end;
ET: \langle Parse XML end tag 667 \rangle; othercases ErrorRecovery(9, [LT, ET]); endcases; end;
```

178 XML PARSER Mizar Parser §666

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
(\S657).
  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 Get contents of XML start tag 666\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 665.
667. \langle \text{ Parse XML end tag } 667 \rangle \equiv
  begin nState \leftarrow eEnd; GetToken; xml\_match(ID)(8, [LT, ET]); OpenStartTaq; GetToken;
  xml\_match(GT)(7, [LT, ET]); GetToken
  end
This code is used in section 665.
668. 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.
\langle \text{Implementation of XML Parser 650} \rangle + \equiv
    { 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; \{ \text{ sometimes we need this } \}
  LT: \langle \text{Parse start of XML tag 669} \rangle;
  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;
```

 $\S669$ Mizar Parser XML PARSER 179

669. 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 669} \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 668.
670. We will want assertions reflecting the parser is in a "start" state or an "end" state.
\langle \text{Implementation of XML Parser } 650 \rangle + \equiv
procedure XMLParserObj.AcceptEndState;
  begin NextElementState; MizAssert(errElRedundant, nState = eEnd);
procedure XMLParserObj.AcceptStartState;
  begin NextElementState; MizAssert(errElMissing, nState = eStart);
  end;
671. \langle Implementation of XML Parser 650\rangle + \equiv
procedure XMLParserObj.OpenStartTag;
  begin nElName \leftarrow nSpelling; nAttrVals.FreeAll;
  end;
672. 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;
```

180 XML PARSER Mizar Parser $\S673$

673. 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 SetAttributeValue.

```
 \begin{array}{l} \textbf{procedure} \ XMLParserObj.ProcessAttributeName; \\ \textbf{begin} \ nAttrVals.Insert(new(XMLAttrPtr,Init(nSpelling, ``))); \\ \textbf{end;} \\ \\ \textbf{procedure} \ XMLParserObj.ProcessAttributeValue; \\ \textbf{begin} \ SetAttributeValue(nSpelling); \\ \textbf{end;} \\ \\ \textbf{procedure} \ XMLParserObj.SetAttributeValue(\textbf{const} \ aVal: \ string); \\ \textbf{begin} \ \textbf{with} \ nAttrVals \ \textbf{do} \ XMLAttrPtr(Items\uparrow[Count-1])\uparrow.nValue \leftarrow aVal; \\ \textbf{end;} \\ \end{array}
```

 $\S674$ Mizar Parser I/O WITH XML 181

File 14

This code is used in section 674.

I/O with XML

674. 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 } 674 \rangle \equiv
  ⟨GNU License 4⟩
unit xml_inout;
  interface
  uses errhan, mobjects, xml_parser;
  \langle Type declarations for XML I/O 675\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;
(Implementation for I/O of XML 676)
end .
675.
       There are only 4 types of streams we care about: Streams, Text Streams, XML Input Streams, and
XML Output Streams.
\langle Type declarations for XML I/O 675\rangle \equiv
  ⟨ Public interface for XML Input Stream 688⟩;
  ⟨Public declaration for Stream Object 680⟩;
  (Public declaration for Text Stream Object 684);
  ⟨ Public declaration for XML Output Stream 693⟩;
```

182 I/O WITH XML Mizar Parser $\S676$

676. 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.

```
⟨ Implementation for I/O of XML 676⟩ ≡ function QuoteStrForXML(const\ aStr:\ string):\ string;
const ValidCharTable = ([`a`...`z`, `A`...`Z`, `0`...`9`, `-`, `\...`, `, `, `, `, `, `, `, `, `, `]`, `, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`, `[`, `]`,
```

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

```
function XMLToStr(\mathbf{const}\ aXMLStr:\ string): string;

\mathbf{var}\ i,h:\ integer;\ lHexNr:\ string;

\mathbf{begin}\ result \leftarrow aXMLStr;

\mathbf{for}\ i \leftarrow length(result) - 5\ \mathbf{downto}\ 1\ \mathbf{do}

\mathbf{begin}\ \langle \operatorname{Transform}\ XML\ entity\ into\ character,\ if\ encountering\ an\ XML\ entity\ at\ i\ 678\ \rangle;

\mathbf{end};

result \leftarrow Trim(result);

\mathbf{end};
```

678. 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 " $\verb"0x"$ " 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 678⟩ ≡ if (result[i] = ^*&^*) \wedge (length(result) \geq i + 5) then begin if (result[i+1] = ^*\#^*) \wedge (result[i+2] = ^*x^*) then begin lHexNr \leftarrow result[i+3] + result[i+4]; h \leftarrow StrToInt(^*Ox^* + lHexNr); Delete(result, i, 5); result[i] \leftarrow chr(h); end; end
```

This code is used in section 677.

This code is used in section 674.

 $\S679$ Mizar Parser I/O WITH XML 183

679. 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 676\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 + \text{`\"'};
     '&': result \leftarrow result + \text{`\&'};
     <: result \leftarrow result + `&lt;`;
     \rightarrow: result \leftarrow result + `>`;
     othercases if integer(aStr[i]) > 127 then result \leftarrow result + `k\#x` + IntToHex(Ord(aStr[i]), 2) + `;`
       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 680} \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 675.
681. We will have a wrapper function for conveniently reporting errors.
\langle \text{Implementation for I/O of XML 676} \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
682.
buffer size to zero, and the buffor 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;
683. Destructor. We close the file, and free up the file buffer.
destructor StreamObj.Done;
  begin Close(nFile); dispose(fFileBuff);
  end;
```

184 I/O WITH XML Mizar Parser §684

Text Stream Object. A text stream is very similar to a Stream Object, except it is specifically for

684.

```
text.
\langle Public declaration for Text Stream Object 684\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 675.
685. We have the convenience function for reporting errors.
\langle \text{Implementation for I/O of XML 676} \rangle + \equiv
procedure TXTStreamObj.Error(Code, Info: integer);
  begin RunError(2000 + Code);
  end;
       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;
687. 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 (§661). 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 688} \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 675.
689. Constructor. The non-debugging code just invokes the XML Parser's constructor (§662).
\langle \text{Implementation for I/O of XML 676} \rangle + \equiv
constructor XMLInStreamObj.OpenFile(const AFileName: string);
  mdebug; write(InfoFile, AFileName); end_mdebug;
  InitParsing(AFileName);
  mdebug ; WriteLn(InfoFile, `\_\reset'); end_mdebug;
  end;
```

 $\S690$ Mizar Parser I/O WITH XML 185

690. 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;
```

691. 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(const aAttrName: string): string; var lAtt: XMLAttrPtr; begin lAtt ← XMLAttrPtr(nAttrVals.ObjectOf(aAttrName)); if Latt ≠ nil then begin GetAttr ← lAtt↑.nValue; exit; end; MizAssert(errMissingXMLAttribute, false);
```

end:

692. 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">"."

```
{ get integer denoted by required XML attribute aAttrName } function XMLInStreamObj.GetIntAttr(\mathbf{const}\ aAttrName:\ string):\ integer; var lInt,\ ec:\ integer; begin val(GetAttr(aAttrName),\ lInt,\ ec);\ GetIntAttr \leftarrow lInt; end;
```

186 I/O WITH XML Mizar Parser §693

693. 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 693\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 675.
      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 676} \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;
```

 $\S696$ Mizar Parser I/O WITH XML 187

```
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;
697. 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:
       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 699 <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 699\rangle \equiv
  if fBuffInd = InOutFileBuffSize then
    begin BlockWrite(nFile, fFileBuff \uparrow, InOutFileBuffSize, fBuffCount); fBuffInd <math>\leftarrow 0;
    end
This code is used in section 698.
700. Print a newline ("\n") to the XML output stream.
\langle \text{Implementation for I/O of XML 676} \rangle + \equiv
procedure XMLOutStreamObj.OutNewLine;
  begin OutChar(#10);
  end:
701. 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;
702. Printing nIndent spaces ("_{\sqcup}") to the output buffer.
{ print nIndent spaces }
procedure XMLOutStreamObj.OutIndent;
  var i: integer:
  begin for i \leftarrow 1 to nIndent do OutChar(``\');
  end:
```

188 I/O WITH XML Mizar Parser §703

703. 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;
```

704. 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;
```

705. 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;
```

706. 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:
```

707. 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;
```

708. 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;
```

709. 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;
```

 $\S710$ Mizar Parser I/O WITH XML 189

710. 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;
```

711. We can now just compose writing the start of a tag ($\S703$), followed by its attributes ($\S712$), and then close the empty-element tag ($\S706$).

```
procedure XMLOutStreamObj.Out_XElWithPos(const fEl: string; const fPos: Position);
begin Out_XElStart(fEl); Out_PosAsAttrs(fPos); Out_XElEndO;
end;
```

712. 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:
```

713. We print a quoted attribute, leveraging printing attributes out to the file ($\S709$). We just need to escape the XML string ($\S676$).

```
 \begin{array}{ll} \textbf{procedure} \ XMLOutStreamObj.Out\_XQuotedAttr(\textbf{const} \ fAt, fVal: \ string); \\ \textbf{begin} \ Out\_XAttr(fAt, QuoteStrForXML(fVal)); \\ \textbf{end}; \end{array}
```

File 15

Vocabulary file dictionaries

```
714. Mizar works with vocabulary files (suffixed with .voc) for introducing new identifiers.
\langle \text{ dicthan.pas } 714 \rangle \equiv
  ⟨GNU License 4⟩
unit dicthan;
  interface
  uses mobjects;
  (Public constants for dicthan.pas 715)
  type SymbolCounters = array ['A' .. 'Z'] of word;
    SymbolIntSeqArr = array ['A' .. 'Z'] of IntSequence;
     (Class declarations for dicthan.pas 716)
     (Public function declarations for dicthan.pas 717)
  implementation
  uses mizenv, xml_inout, xml_dict;
  ⟨Implementation for dicthan.pas 718⟩
  end.
      We recall from Adam Grabowski, Artur Korniłowicz, and Adam Naumowicz's "Mizar in a Nutshell"
(\S4.3, 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 } 715 \rangle \equiv
const
  StandardPriority = 64;
  AvailableSymbols = [ `G`, `K`, `L`, `M`, `O`, `R`, `U`, `V`];
This code is used in section 714.
       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 } 716 \rangle \equiv
  \langle \text{Symbol for vocabulary } 722 \rangle;
  ⟨Abstract vocabulary object declaration 731⟩;
  ⟨Vocabulary object declaration 733⟩;
This code is used in section 714.
```

```
⟨Public function declarations for dicthan.pas 717⟩ ≡
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 714.
       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 } 718 \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);
  \langle Initialize lKind, but exit if dictionary line contains invalid symbol 719\rangle;
  delete\_prefix;
  case lKind of
  '0': (Check if functor symbol is valid 720);
  R: (Check if predicate symbol is valid 721);
  othercases begin if Pos(` \_ `, lLine) > 0 then exit;
    IsValidSymbol \leftarrow true;
    end;
  endcases;
  end:
See also sections 723, 727, 732, 734, 738, 740, and 741.
This code is used in section 714.
719. 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 \text{Initialize } lKind, \text{ but exit if dictionary line contains invalid symbol } 719 \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 718.
```

720. 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 720⟩ ≡ 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
This code is used in section 718.
```

721. 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.

```
 \begin{split} &\langle \text{Check if predicate symbol is valid } 721 \rangle \equiv \\ & \mathbf{begin} \ lPos \leftarrow Pos(`\_`, lLine); \\ & \mathbf{if} \ lPos \neq 0 \ \mathbf{then} \quad \{ \ lLine \ \text{contains a space} \} \\ & \mathbf{begin} \ lLine \leftarrow TrimString(Copy(lLine, lPos, length(lLine))); \\ & \mathbf{if} \ Pos(`\_`, lLine) > 0 \ \mathbf{then} \ \ exit; \\ & \mathbf{end}; \\ & \mathit{IsValidSymbol} \leftarrow \mathit{true}; \\ & \mathbf{end} \end{split}
```

This code is used in section 718.

This code is used in section 716.

722. TSymbol. These are used in kernel/accdict.pas. The *Kind* is its one-letter kind (discussed in §715), and *Repr* is its lexeme. For functors, its priority is stored as its *Prior*.

The "infinitive" appears to be only used for predicates.

```
 \langle \text{Symbol for vocabulary } 722 \rangle \equiv \\ PSymbol = \uparrow TSymbol; \\ TSymbol = \mathbf{object} \; (MObject) \\ Kind: \; char; \\ Repr, Infinitive: \; string; \\ Prior: \; byte; \\ \mathbf{constructor} \; Init(fKind: char; fRepr, fInfinitive: string; fPriority: byte); \\ \mathbf{constructor} \; Extract(\mathbf{const} \; aLine: \; string); \\ \mathbf{function} \; SymbolStr: \; string; \\ \mathbf{constructor} \; Load(\mathbf{var} \; aText: text); \\ \mathbf{procedure} \; Store(\mathbf{var} \; aText: text); \\ \mathbf{destructor} \; Done; \; virtual; \\ \mathbf{end}
```

723. 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 718\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;
```

724. 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
`O`:\ begin\ lPos \leftarrow Pos(`\_`,Repr);\ Prior \leftarrow StandardPriority;
if lPos \neq 0 then \langle Initialize explicit priority for functor entry in dictionary 726\rangle;
end;
`R`:\ begin\ lPos \leftarrow Pos(`\_`,Repr);
if lPos \neq 0 then \langle Initialize explicit infinitive for a predicate entry in dictionary 725\rangle;
end;
endcases;
end;
```

725. 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 $725\rangle \equiv$

```
begin lRepr \leftarrow Repr; Repr \leftarrow   ; Repr \leftarrow TrimString(Copy(lRepr, 1, lPos - 1)); Infinitive \leftarrow TrimString(Copy(lRepr, lPos + 1, length(lRepr))); end
```

This code is used in section 724.

726. Functors with explicit priorities require parsing that priority. It is assumed that a single whitespace separates the lexeme from the priority.

```
\langle Initialize explicit priority for functor entry in dictionary 726 \rangle \equiv begin lRepr \leftarrow Repr; Repr \leftarrow ``; val(TrimString(Copy(lRepr, lPos + 1, length(lRepr))), Prior, lCode); {Store the priority} Repr \leftarrow TrimString(Copy(lRepr, 1, lPos - 1)); {Store the lexeme} end
```

This code is used in section 724.

Mizar Parser

194

727. 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 } 718 \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;
      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 (\S718), 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);
  end;
729.
       Storing a TSymbol in a file amounts to writing its serialization (§727) to the file.
procedure TSymbol.Store(var aText : text);
  begin WriteLn(aText, SymbolStr);
  end;
730. Destructor. We just reset the lexeme and infinitive strings to be empty strings.
destructor TSymbol.Done;
  begin Repr \leftarrow ``; Infinitive \leftarrow ``;
  end:
       Abstract vocabulary objects. This is used in kernel/impobjs.pas. We recall (§714) that the
Symbol Counters are just an enumerated type consisting of a single uppercase Latin Letter.
\langle Abstract vocabulary object declaration 731 \rangle \equiv
  AbsVocabularyPtr = \uparrow AbsVocabularyObj;
  AbsVocabularyObj = \mathbf{object} \ (MObject)
     fSymbolCnt: SymbolCounters;
```

This code is used in section 716.

end

constructor *Init*;

destructor Done: virtual:

```
We only have the constructor and destructor for abstract vocabulary objects.
\langle \text{Implementation for dicthan.pas } 718 \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 733 \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 716.
734. Constructor (Empty vocabulary). We can construct the empty vocabulary by just initializing
the underlying collection.
\langle \text{Implementation for dicthan.pas } 718 \rangle + \equiv
constructor TVocabulary.Init;
  begin FillChar(fSymbolCnt, SizeOf(fSymbolCnt), 0); Reprs.Init(10, 10);
  end;
735. 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 737\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 737 \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 736.

196

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 } 718 \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 739);
  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 } 739 \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 738.

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 \text{Implementation for dicthan.pas } 718 \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;
```

741. Miscellaneous public-facing functions.

```
 \langle \text{Implementation for dicthan.pas } 718 \rangle + \equiv \\ \text{function } GetPrivateVoc(\textbf{const } fName: string) : PVocabulary; \\ \text{var } lName: string; \\ \text{begin } lName \leftarrow fName; \\ \text{if } ExtractFileExt(lName) = \text{``then } lName \leftarrow lName + \text{`.voc'}; \\ \text{if } \neg MFileExists(lName) \text{ then} \\ \text{begin } GetPrivateVoc \leftarrow \text{nil}; exit; \\ \text{end}; \\ GetPrivateVoc \leftarrow new(PVocabulary, ReadPrivateVoc(lName)); \\ \text{end}; \\ \\ \text{end}; \\ \end{aligned}
```

742. 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 (§738).

```
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;
```

743. 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;
```

744. Storing a vocabulary delegates much work ($\S740$). 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;
```

198

Like StoreMmlVcb, this function is not used anywhere in Mizar. This appears to produce the XMLequivalent 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 746⟩;
         ⟨ Write symbols to vocabulary XML file 747⟩;
         VCXfile.Out\_XElEnd(XMLElemName[elVocabulary]);
  VCXfile.Out\_XElEnd(XMLElemName[elVocabularies]); dispose(VCXfile,Done);
  end;
746. We write out the counts of each kind of definition appearing in the article.
\langle Write vocabulary counts to XML file 746\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 745.
747. We write out each symbol appearing in the article's vocabulary.
\langle Write symbols to vocabulary XML file 747\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:
  VCXfile.Out\_XElEnd(XMLElemName[elSymbols])
This code is used in section 745.
```

§748 Mizar Parser SCANNER 199

File 16

Scanner

748. 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.

```
 \begin{array}{l} \langle \, {\rm scanner.pas} \,\, 748 \, \rangle \equiv \\ \langle \, {\rm GNU \,\, License} \,\, 4 \, \rangle \\ \\ {\rm unit \,\,} scanner; \\ {\rm interface} \,\, ; \\ {\rm uses \,\,} errhan, mobjects; \\ {\rm const} \,\, MaxLineLength = 80; \\ MaxConstInt = 2147483647; \quad \big\{ = 2^{31} - 1, \, {\rm maximal \,\, signed \,\,} 32\text{-bit integer} \, \big\} \\ \langle \, {\rm Type \,\,} declarations \,\, for \,\, scanner \,\, 749 \, \rangle \\ {\rm implementation} \\ {\rm uses \,\,} mizenv, librenv, mconsole, xml\_dict, xml\_inout; \\ \langle \, {\rm Implementation \,\,} for \,\, scanner.pas \,\, 750 \, \rangle \\ {\rm end} \,\, . \\ {\rm See \,\,} also \,\, section \,\, 879. \end{array}
```

749. 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 749⟩ ≡

type ASCIIArr = array [chr(0) .. chr(255)] of byte;

LexemRec = record Kind: char;

Nr: integer;
end;
⟨Token object class 751⟩;
⟨Tokens collection class 753⟩;
⟨MToken object class 761⟩;
⟨MTokeniser class 764⟩;
⟨MScanner object class 792⟩;

This code is used in section 748.
```

200 SCANNER Mizar Parser §750

750. 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 750 \rangle \equiv
\mathbf{var}\ DefaultAllowed\colon AsciiArr =
 { '_' allowed in identifiers by default! }
 See also sections 752, 754, 755, 756, 759, 760, 762, 763, 765, 768, 769, 773, 788, 789, 791, 793, 802, 803, 804, 805, and 806.
This code is used in section 748.
    Tokens object. A token contains a lexeme, but it extends an MStr object.
\langle \text{ Token object class } 751 \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 749.
    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 750 \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;
```

§753 Mizar Parser COLLECTIONS OF TOKENS 201

Section 16.1. COLLECTIONS OF TOKENS

else begin $CollectToken \leftarrow true; Insert(lToken)$

 $\mathbf{end};$

753. 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 753\rangle \equiv
  TokensCollection = \mathbf{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 749.
       Construct empty token collection.
\langle Implementation for scanner.pas 750 \rangle + \equiv
constructor TokensCollection.InitTokens;
  begin Init(100);
  end;
755. 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 750 \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)
```

756. 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 750\rangle += constructor TokensCollection.LoadDct(const aDctFileName: string); var Dct: text; lKind, lDummy: AnsiChar; lNr: integer; lString: string; i: integer; c: char; begin assign(Dct, aDctFileName + ´.dct´); reset(Dct); InitTokens; \langle Load all tokens from the dictionary 757\rangle; close(Dct); \langle Index first character appearances among definitions 758\rangle; end;
```

757. We just iterate through the dictionary, constructing a new token for each line we read.

```
\langle \text{Load all tokens from the dictionary 757} \rangle \equiv 
while \neg seekEof(Dct) do
begin readln(Dct, lKind, lNr, lDummy, lString);
Insert(new(TokenPtr, Init(char(lKind), lNr, lString)));
end
```

This code is used in section 756.

758. We index the first appearance of each leading character in a token.

```
\langle Index first character appearances among definitions 758\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 756.

759. 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.

```
⟨ Implementation for scanner.pas 750⟩ +≡
procedure TokensCollection.SaveDct(const aDctFileName: string);
var i: integer; DctFile: text;
begin assign(DctFile, aDctFileName + ´.dct´); rewrite(DctFile);
for i ← 0 to Count − 1 do
    with TokenPtr(Items↑[i])↑, fLexem do writeln(DctFile, AnsiChar(Kind), Nr, ´u´, fStr);
close(DctFile);
end;
```

 $\S760$ Mizar Parser COLLECTIONS OF TOKENS 203

760. 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 750 \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;
```

204 MIZAR TOKEN OBJECTS Mizar Parser §761

Section 16.2. MIZAR TOKEN OBJECTS

```
761. This appears to be tokens for a specific file. An MToken extends a Token (§751).
⟨MToken object class 761⟩ ≡

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 749.
```

762. 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 750\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;
```

763. 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 } 750 \rangle + \equiv 
const Numeral = \text{`N'}; Identifier = \text{`I'}; ErrorSymbol = \text{`?'}; EOT = \text{`!'};
```

8764 Mizar Parser TOKENISER 205

Section 16.3. TOKENISER

764. The first step in lexical analysis is to transform a character stream into a token stream. The Tokeniser extends the MToken object ($\S761$), which in turn extends the Token object ($\S751$).

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 764 \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 749.

765. Spelling boils down to three cases (c.f., types of tokens §763): 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 750⟩ +≡
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 766⟩
   else ⟨Spell an error or EOF for the MTokeniser 767⟩;
end;
```

206 TOKENISER Mizar Parser §766

766. 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.

```
⟨ Spell an identifier for the MTokeniser 766 ⟩ ≡ 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; end
```

This code is used in section 765.

767. 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 767⟩ ≡ 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 765.

768. Constructor. Initialising a tokeniser starts with a blank phrase and kind, with most fields set to zero.

```
⟨ Implementation for scanner.pas 750⟩ +≡ 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;
```

769. Destructor. This chains to free up several fields, just invoking their destructors.

```
⟨Implementation for scanner.pas 750⟩ +≡
destructor MTokeniser.Done;
begin fPhrase ← ´´; fTokensBuf.Done; fTokens.Done; fIdents.Done;
end;
```

770. 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 (§797);
- (2) all other invalid characters are replaced with the unit_separator character (c.f., §798).

```
define record\_separator \equiv chr(30)
define unit\_separator \equiv chr(31)
```

771. 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{array}{c|c} \overline{begin_{\sqcup}} \\ \hline theorem_{\sqcup} \\ \hline \hline for_{\sqcup} x_{\sqcup} being_{\sqcup} object_{\sqcup} \\ \hline holds_{\sqcup} x=_{\sqcup} x;_{\sqcup} \\ \end{array}
```

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.

208 TOKENISER Mizar Parser $\S772$

772. 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 (§789).

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 (§763): 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 (§779) in this procedure.

```
\langle \text{ Variables for slicing a phrase } 772 \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 775, 778, 781, 783, and 785.
This code is used in section 773.
       \langle Implementation for scanner.pas 750 \rangle + \equiv
procedure MTokeniser.SliceIt;
  var \( \text{Variables for slicing a phrase 772} \)
  begin MizAssert(2333, fTokensBuf.Count = 0); { Requires: token buffer is empty }
  lCurrChar \leftarrow 1; \ lPos \leftarrow fPhrasePos;
  \langle \text{Slice pragmas } 774 \rangle;
  while fPhrase[lCurrChar] \neq 1 do
     begin (Determine the ID 776);
     ⟨Try to find a dictionary symbol 779⟩;
     if EndOfSymbol < EndOfIdent then \langle Check identifier is not a number 782\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 787);
     end;
  end;
774. We begin by slicing pragmas. This will insert the pragma into the tokens buffer.
  Note that the "$EOF" pragma indicates that we should treat the file as ending here. So we comply with
the request, inserting the EOT (end of text) token as the next token to be offered to the user.
\langle \text{Slice pragmas } 774 \rangle \equiv
  if (lPos.Col = 1) \land (Pos("::$", 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 773.

 $\S775$ Mizar Parser TOKENISER 209

775. 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., §798). 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.

```
⟨ Variables for slicing a phrase 772⟩ +≡
IdentLength: integer;

776. ⟨ Determine the ID 776⟩ ≡
{ 1. attempt to determine the ID }
EndOfIdent ← lCurrChar;
if IsIdentifierFirstLetter(fPhrase[EndOfIdent]) then
    while (EndOfIdent < length(fPhrase)) ∧ IsIdentifierLetter(fPhrase[EndOfIdent]) do
    inc(EndOfIdent);
IdentLength ← EndOfIdent - lCurrChar;
if fPhrase[EndOfIdent] ≤ unit_separator then
    ⟨ Whoops! ID turns out to be invalid, insert an error token, then continue 777⟩;
dec(EndOfIdent)</pre>
This code is used in section 773.
```

777. Recall (§797), 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 777 \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 776.

210 TOKENISER Mizar Parser $\S778$

778. 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 ($\S 387$), sorted lists have a field FIndex which is an array of indices (which are sorted while leaving the underlying array FIndex 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(\#) \equiv Items \uparrow [fIndex \uparrow [\#]]
  define the\_token(\#) \equiv TokenPtr(the\_item(\#))\uparrow
\langle \text{ Variables for slicing a phrase } 772 \rangle + \equiv
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}
779. Reserved keywords and defined terms are loaded into the fTokens dictionary.
\langle \text{Try to find a dictionary symbol } 779 \rangle \equiv
  EndOfPhrase \leftarrow lCurrChar; FoundToken \leftarrow nil; EndOfSymbol \leftarrow EndOfPhrase - 1;
       { initialized for comparison }
  lToken \leftarrow fTokens.fFirstChar[fPhrase[EndOfPhrase]]; inc(EndOfPhrase);
  if (lToken > 0) then
     with fTokens do
       begin lIndex \leftarrow 2;
       repeat { infinite loop }
          (If we matched a dictionary entry, then initialize FoundToken 780);
         if fPhrase[EndOfPhrase] = ` ' ' then break; { we are done! }
         if (lIndex < length(the\_token(lToken).fStr)) \land
                 (the\_token(lToken).fStr[lIndex] = fPhrase[EndOfPhrase]) then
            begin inc(lIndex); inc(EndOfPhrase) end { iterate, look at next character }
         else if (lToken < Count - 1) then \{ try looking for the last matching item \}
              begin if (copy(the\_token(lToken).fStr, 1, lIndex - 1) =
                      copy(the\_token(lToken + 1).fStr, 1, lIndex - 1)) then inc(lToken) { iterate }
              else break; { we are done! }
              end
          else break; { we are done! }
       until false;
       end
```

This code is used in section 773.

 $\S780$ Mizar Parser TOKENISER 211

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 780 $\rangle \equiv$ 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 779. When the identifier is not a number, we insert an "identifier" token into the tokens buffer. $\langle \text{ Variables for slicing a phrase } 772 \rangle + \equiv$ lFailed: integer; { index of first non-digit character } I: integer; { index ranging over the raw lexeme string } lSpelling: string; { raw lexeme as a string } **782.** \langle Check identifier is not a number $782 \rangle \equiv$ **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! 786); (Add token to tokens buffer and iterate 784); This code is used in section 773. We add an *Identifier* token to the tokens buffer. $\langle \text{ Variables for slicing a phrase } 772 \rangle + \equiv$ lIdent: TokenPtr; **784.** \langle Add token to tokens buffer and iterate $784 \rangle \equiv$ $lIdent \leftarrow new(TokenPtr, Init(Identifier, fIdents.Count + 1, lSpelling));$ 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 782.

212 TOKENISER Mizar Parser $\S785$

785. 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, §748), 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 } 772 \rangle + \equiv
lNumber: longint;
J: integer;
786. Whoops! Identifier turned out to be a number! 786 \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 782.
787. 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 787 \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 773.
       We have purely abstract methods which will invoke Abstract1 (\S 308), which raises a runtime error.
\langle \text{Implementation for scanner.pas } 750 \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;
```

§789 Mizar Parser TOKENISER 213

789. 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 (§793) and SliceIt (§772) do the actual work.

```
⟨ Implementation for scanner.pas 750⟩ +≡
procedure MTokeniser.GetToken;
begin if fLexem.Kind = EOT then exit;
if fTokensBuf.Count > 0 then
   begin ⟨ Pop a token from the underlying tokens stack 790⟩;
   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 790⟩;
end;
```

790. 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 } 790 \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 789 and 789.
```

791. Testing if the given character is an identifier character or not requires invoking the abstract method IsIdentifierLetter (§788).

```
\langle Implementation for scanner.pas 750\rangle +\equiv function MTokeniser.IsIdentifierFirstLetter(ch: char): boolean; begin IsIdentifierFirstLetter \leftarrow IsIdentifierLetter(ch); end:
```

214 SCANNER OBJECT Mizar Parser §792

Section 16.4. SCANNER OBJECT

This code is used in section 749.

```
This extends the Tokeniser class (§764). It is the only class extending the Tokeniser class.
\langle MScanner object class 792 \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
```

793. 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 750⟩ +≡
procedure MScannObj.GetPhrase;
const Prohibited: ASCIIArr = ⟨ Characters prohibited by MScanner 794⟩;
var i,k: integer;
begin fPhrasePos.Col ← fPhrasePos.Col + length(fPhrase) - 1;
⟨ Find the first significant 'sign' 795⟩;
for i ← fPhrasePos.Col to length(fCurrentLine) do
    if fCurrentLine[i] = ´ □ ´ then break;
fPhrase ← Copy(fCurrentLine, fPhrasePos.Col, i - fPhrasePos.Col + 1);
end;
```

§794 Mizar Parser SCANNER OBJECT 215

794. 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 793.

795. Note that the *fCurrentLine* will end with a whitespace, when we have not consumed the entire underlying input stream.

```
⟨ Find the first significant 'sign' 795⟩ ≡
while fCurrentLine[fPhrasePos.Col] = ´¬¬ do
begin if fPhrasePos.Col ≥ length(fCurrentLine) then ⟨ Populate the current line 796⟩;
inc(fPhrasePos.Col);
end
```

This code is used in section 793.

796. 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 796⟩ ≡

begin if EndOfText then exit;

inc(fPos.Line); inc(fPhrasePos.Line); readln(fSourceFile, fCurrentLine);

⟨ Scan for pragmas, and exit if we found one 800⟩;

⟨ Skip comments 801⟩;

⟨ Trim whitespace from the right of the current line 799⟩;

⟨ Replace every invalid character in current line with the unit character 798⟩;

fCurrentLine \leftarrow fCurrentLine + `\_`;

if \neg LongLines then

if length(fCurrentLine) > MaxLineLength then ⟨ Replace end of long line with record separator 797⟩;

{ Assert: we have fCurrentLine end in "\_"⟩

fPhrasePos.Col \leftarrow 0; fPos.Col \leftarrow 0;
end
```

This code is used in section 795.

216 SCANNER OBJECT Mizar Parser §797

797. 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 \mbox{ Replace end of long line with record separator } 797 \rangle \equiv \\ \mbox{ begin } delete(fCurrentLine, MaxLineLength + 1, length(fCurrentLine)); } \\ fCurrentLine[MaxLineLength - 1] \leftarrow record\_separator; \\ fCurrentLine[MaxLineLength] \leftarrow `\ '\ '\ '; \\ \mbox{ end } \\ \end{cases}
```

This code is used in section 796.

798. 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 798} \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 796.
```

799. We will trim whitespace from the right of the current line at least twice.

```
 \begin{split} &\langle \operatorname{Trim} \ \operatorname{whitespace} \ \operatorname{from} \ \operatorname{the} \ \operatorname{right} \ \operatorname{of} \ \operatorname{the} \ \operatorname{current} \ \operatorname{line} \ 799 \,\rangle \equiv \\ &k \leftarrow length(fCurrentLine); \\ &\mathbf{while} \ (k > 0) \land (fCurrentLine[k] = ` \sqcup `) \ \mathbf{do} \ \operatorname{dec}(k); \\ &\operatorname{delete} (fCurrentLine, k + 1, length(fCurrentLine)) \end{split}
```

This code is used in sections 796 and 800.

800. 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 (§802).

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 800⟩ ≡ 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 799⟩; fCurrentLine \leftarrow fCurrentLine + `\_\'; fPhrase \leftarrow Copy(fCurrentLine,1,length(fCurrentLine)); <math>fPhrasePos.Col \leftarrow 1; fPos.Col \leftarrow 0; exit end
```

801. 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 (§802).

```
\langle \text{Skip comments 801} \rangle \equiv k \leftarrow Pos(\ ::\ :\ fCurrentLine); \ \ \{ \text{Comment} \} 
if (k \neq 0) then
begin ProcessComment(fPhrasePos.Line, k, copy(fCurrentLine, k, length(fCurrentLine)));
delete(fCurrentLine, k + 1, length(fCurrentLine)); \ fCurrentLine[k] \leftarrow \ :\ :
```

This code is used in section 796.

This code is used in section 796.

 $\S802$ Mizar Parser SCANNER OBJECT 217

802. "Processing a comment" really means skipping the comment.

```
⟨Implementation for scanner.pas 750⟩ +≡ procedure MScannObj.ProcessComment(fLine, fStart: integer; cmt: string); begin end;
```

803. 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 } 750 \rangle + \equiv function MScannObj.EndOfText: boolean; begin EndOfText \leftarrow (fPhrasePos.Col \geq length(fCurrentLine)) \wedge eof(fSourceFile); end;
```

804. Testing if a character is an identifier letter amounts to testing if it is allowed (i.e., not disallowed).

```
\langle Implementation for scanner.pas 750\rangle +\equiv function MScannObj.IsIdentifierLetter(ch:char): boolean; begin <math>IsIdentifierLetter \leftarrow Allowed[ch] \neq 0; end;
```

805. 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 §756). The buffer size for reading aFileName is initially #4000.

CAUTION: This will cause a memory leak if you try to do unit testing with the parser. Specifically the *loadDct* method appears to allocate memory which is never freed adequately. I worry this might be sympotmatic of a larger problem.

```
⟨ Implementation for scanner.pas 750⟩ +≡
constructor MScannObj.InitScanning(const aFileName, aDctFileName: string);
begin inheritedInit; Allowed ← DefaultAllowed; fTokens.LoadDct(aDctFileName); { memory leaked }
assign(fSourceFile, aFileName); fSourceBuffSize ← #4000; getmem(fSourceBuff, fSourceBuffSize);
settextbuf(fSourceFile, fSourceBuff↑, #4000); reset(fSourceFile); fCurrentLine ← ´¬¬; GetToken;
end;
```

806. Destructor. We must remember to close the source file, free the buffer, close the lights, and lock the doors.

```
⟨ Implementation for scanner.pas 750⟩ +≡
destructor MScannObj.Done;
begin close(fSourceFile); FreeMem(fSourceBuff,fSourceBuffSize); fCurrentLine ← ´´; inheritedDone;
end;
```

218 FORMAT Mizar Parser §807

File 17

Format

807. 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/1983) 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 807 \rangle \equiv
  ⟨GNU License 4⟩
unit _formats;
  interface
  uses mobjects, scanner, dicthan, xml_inout;
    ⟨ Declare classes for _formats.pas 809⟩
  function CompareFormats(aItem1, aItem2: Pointer): Integer;
  function In_Format(fInFile: XMLInStreamPtr): MFormatPtr;
    ⟨Global variables (_formats.pas) 808⟩
  implementation
  uses errhan, xml_dict, xml_parser
      mdebug , info end_mdebug;
    ⟨Implementation for _formats.pas 810⟩
  end.
808.
\langle \text{Global variables (\_formats.pas) } 808 \rangle \equiv
var gFormatsColl: MFormatsList; gPriority: BinIntFunc; gFormatsBase: integer;
This code is used in section 807.
```

 $\S 809$ Mizar Parser FORMAT 219

809. 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 } 809 \rangle \equiv
  \langle \text{ Declare } MFormat \text{ object } 811 \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 } 813 \rangle;
  \langle \text{ Declare } MInfixFormat \text{ object } 815 \rangle;
  \langle \text{ Declare } MBracketFormat \text{ object } 817 \rangle;
  \langle \text{ Declare } MFormatsList \text{ object } 825 \rangle;
This code is used in section 807.
       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 } 810 \rangle \equiv
   (Constructors for derived format classess 812)
   (Compare formats 819)
   (Implementation for MFormatsList 826)
   (Read formats from an XML input stream 837)
   ⟨Implement MFormatObj 838⟩
This code is used in section 807.
       Format base class. All format instances have a lexeme called its fSymbol. Recall that LexemeRec
(§749) 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 } 811 \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 809.
        The constructor expects the "kind" of the object and its symbol number.
\langle Constructors for derived format classess 812\rangle \equiv
constructor MFormatObj.Init(aKind : Char; aSymNr : integer);
  begin fSymbol.Kind \leftarrow aKind; fSymbol.Nr \leftarrow aSymNr;
  end:
See also sections 814, 816, and 818.
```

This code is used in section 810.

220 FORMAT Mizar Parser §813

```
813. Prefix format object.
\langle \text{ Declare } MPrefixFormat \text{ object } 813 \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 809.
814. Prefix formats track how many arguments are to the right of the prefix symbol.
\langle Constructors for derived format classess 812\rangle + \equiv
constructor MPrefixFormatObj.Init(aKind: Char; aSymNr, aRArqsNbr: integer);
  begin fSymbol.Kind \leftarrow aKind; fSymbol.Nr \leftarrow aSymNr; fRightArgsNbr \leftarrow aRArgsNbr;
  end;
815. Infix format object.
\langle \text{ Declare } MInfixFormat \text{ object } 815 \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 809.
       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 812\rangle + \equiv
constructor MInfixFormatObj.Init(aKind: Char; aSymNr, aLArqsNbr, aRArqsNbr: integer);
  begin fSymbol.Kind \leftarrow aKind; fSymbol.Nr \leftarrow aSymNr; fLeftArgsNbr \leftarrow aLArgsNbr;
  fRightArgsNbr \leftarrow aRArgsNbr;
  end;
817. Bracket format object.
\langle \text{ Declare } MBracketFormat \text{ object } 817 \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 809.
818. \langle Constructors for derived format classess 812\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:
```

 $\S819$ Mizar Parser FORMAT 221

819. 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.

```
or less.
\langle \text{ Compare formats 819} \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 820} \rangle;
  end:
This code is used in section 810.
820. 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 820\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 821)
This code is used in section 819.
       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 821\rangle \equiv
  case MFormatPtr(aItem1)\uparrow.fSymbol.Kind of
  'J', 'U': CompareFormats \leftarrow 0;
  'G', 'L', 'M', 'V': (Compare prefix symbols 822);
  '0', 'R': (Compare infix symbols 824);
  'K': (Compare bracket symbols 823);
  endcases
This code is used in section 820.
       Comparing prefixing symbols, at this points, can only compare the number of arguments to the right.
\langle \text{ Compare prefix symbols 822} \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
       CompareFormats \leftarrow 0
```

This code is used in section 821.

222 FORMAT Mizar Parser §823

823. 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.

 $\langle \text{Compare bracket symbols 823} \rangle \equiv \\ \text{if } \textit{MBracketFormatPtr(aItem1)} \uparrow .fRightSymbolNr < \textit{MBracketFormatPtr(aItem2)} \uparrow .fRightSymbolNr \\ \text{then } \textit{CompareFormats} \leftarrow -1 \\ \text{else if } \textit{MBracketFormatPtr(aItem1)} \uparrow .fRightSymbolNr = \textit{MBracketFormatPtr(aItem2)} \uparrow .fRightSymbolNr \\ \text{then} \\ \text{if } \textit{MBracketFormatPtr(aItem1)} \uparrow .fArgsNbr < \textit{MBracketFormatPtr(aItem2)} \uparrow .fArgsNbr \text{ then} \\ \textit{CompareFormats} \leftarrow -1 \\ \text{else if } \textit{MBracketFormatPtr(aItem1)} \uparrow .fArgsNbr = \textit{MBracketFormatPtr(aItem2)} \uparrow .fArgsNbr \text{ then} \\ \langle \text{Compare infix symbols 824} \rangle$

This code is used in section 821.

824. 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 824\rangle \equiv
```

- if $MInfixFormatPtr(aItem1)\uparrow.fLeftArgsNbr < MInfixFormatPtr(aItem2)\uparrow.fLeftArgsNbr$ 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 821 and 823.

 $\S825$ Mizar Parser LIST OF FORMATS 223

Section 17.1. LIST OF FORMATS

825. 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 } 825 \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, aLArgsNbr, aRArgsNbr: integer): integer;
  end
```

This code is used in section 809.

826. 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 826⟩ ≡ 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 810.

224 LIST OF FORMATS Mizar Parser $\S 827$

827. Looking up an infix functor format (§815). 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;
828. 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;
829. 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;
830.
      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;
```

 $\S 831$ Mizar Parser LIST OF FORMATS 225

831. 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: 833. 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; 834. 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:

226 LIST OF FORMATS Mizar Parser §835

```
Constructor. Construct the empty list of formats.
constructor MFormatsList.Init(ALimit: Integer);
  begin InitSorted(ALimit, CompareFormats);
  end:
836.
       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));
    qPriority.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:
837. We can read exactly one format from an XML input stream.
\langle \text{Read formats from an XML input stream } 837 \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 810.

 $\S 838$ Mizar Parser LIST OF FORMATS 227

```
Conversely, we can print to an output stream an XML representation for a format object.
\langle \text{Implement } MFormatObj | 838 \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 810.
      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;
840. 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:
```

228 SYNTAX Mizar Parser §841

File 18

Syntax

```
841. This describes the syntax for the Mizar language, using expressions, subexpressions, blocks, and
"items" (statements).
  We will need to recall StackedObj from mobjects.pas (§449).
\langle \text{syntax.pas 841} \rangle \equiv
  (GNU License 4)
unit syntax;
  interface
  uses mobjects, errhan; (Interface for syntax.pas 848)
  implementation
  uses mconsole
    mdebug, info end_mdebug;
    (Implementation for syntax.pas 843)
  end.
       The maximum number of "visible" arguments to an expression is set here, at 10.
842.
\langle \text{Public constants for syntax.pas } 842 \rangle \equiv
const MaxVisArqNbr = 10;
This code is used in section 848.
       The implementation for the abstract syntax of Mizar is rather uninteresting, since most of the methods
are abstract.
\langle \text{Implementation for syntax.pas } 843 \rangle \equiv
  (Subexpression constructor 874)
   (Subexpression destructor 875)
   Expression constructor 871
  (Subexpression procedures 878)
   Create a subexpression for an expression 872
   (Item object implementation 864)
  (Block object implementation 854)
  (Public procedures implementation for syntax.pas 844)
This code is used in section 841.
844. 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 844⟩ ≡
procedure KillSubexpression;
  begin if qSubexpPtr = nil then RunTimeError(2144)
  else dispose(gSubexpPtr, Done);
  end;
See also sections 845, 846, and 847.
This code is used in section 843.
```

845 Mizar Parser SYNTAX 229

```
845.
⟨ Public procedures implementation for syntax.pas 844 ⟩ +≡
procedure KillExpression;
  begin if qExpPtr = nil then RunTimeError(2143)
  else dispose(gExpPtr, Done);
  end;
846.
       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 844⟩ +≡
procedure KillItem;
  begin if gItemPtr = nil then RunTimeError(2142)
  else begin gItemPtr \uparrow .Pop; dispose(gItemPtr, Done); end;
  DisplayLine(CurPos.Line, ErrorNbr);
  end;
847.
⟨ Public procedures implementation for syntax.pas 844 ⟩ +≡
procedure KillBlock;
  begin if gBlockPtr = \mathbf{nil} then RunTimeError(2141)
  else begin gBlockPtr\uparrow.Pop; dispose(gBlockPtr,Done);
    end:
  DisplayLine(CurPos.Line, ErrorNbr);
  end;
848.
\langle \text{Interface for syntax.pas } 848 \rangle \equiv
  (Public constants for syntax.pas 842)
type (BlockKinds (syntax.pas) 852)
  ⟨ItemKinds (syntax.pas) 862⟩
  ⟨ExpKinds (syntax.pas) 869⟩
  ⟨Block object interface 853⟩;
  (Class declaration for Item object 863);
  (Subexpression object class 873);
  \langle \text{Expression class declaration } 870 \rangle;
  (Public procedures for syntax.pas 849)
  (Public variables for syntax.pas 850)
This code is used in section 841.
849. \langle \text{Public procedures for syntax.pas 849} \rangle \equiv
procedure KillBlock;
procedure KillItem;
procedure KillExpression;
procedure KillSubexpression;
This code is used in section 848.
```

230 SYNTAX Mizar Parser $\S 850$

850. These global public variables for syntax will be manipulated by the parser.

```
\langle \text{Public variables for syntax.pas } 850 \rangle \equiv  var gBlockPtr: BlockPtr = nil; gItemPtr: ItemPtr = nil; gExpPtr: ExpressionPtr = nil; gSubexpPtr: SubexpPtr = nil; This code is used in section 848.
```

§851 Mizar Parser BLOCK OBJECT 231

Section 18.1. BLOCK OBJECT

851. 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. 7. UML class diagram for Block object class.

852. There are about a dozen different kinds of blocks.

```
 \langle \ BlockKinds \ (\texttt{syntax.pas}) \ \$52 \ \rangle \equiv \\ BlockKind = (blMain, blDiffuse, blHereby, blProof, blDefinition, blNotation, blRegistration, blCase, \\ blSuppose, blPublicScheme);
```

This code is used in section 848.

```
853. \langle Block object interface 853\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 848.

854. The constructor for a Block will initialize its Previous pointer to point at the global gBlockPtr instance.

```
⟨ Block object implementation 854⟩ ≡ constructor BlockObj.Init(fBlockKind : BlockKind); begin nBlockKind ← fBlockKind; Previous ← gBlockPtr; end;
See also sections 855, 856, 857, 858, 859, and 860.
This code is used in section 843.
```

232 BLOCK OBJECT Mizar Parser §855

```
855.
      Note that popping a block object is left for subclasses to handle.
\langle Block object implementation 854\rangle + \equiv
procedure BlockObj.Pop;
  begin end;
856. \langle Block object implementation 854 \rangle + \equiv
destructor BlockObj.Done;
  begin gBlockPtr \leftarrow BlockPtr(Previous);
  end;
857. Abstract methods.
\langle Block object implementation 854\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;
858. \langle Block object implementation 854 \rangle + \equiv
procedure BlockObj.CreateItem(fItemKind: ItemKind);
  begin gItemPtr \leftarrow new(ItemPtr, Init(fItemKind));
  end;
      \langle Block object implementation 854\rangle + \equiv
procedure BlockObj.CreateBlock(fBlockKind: BlockKind);
  begin gBlockPtr \leftarrow new(BlockPtr, Init(fBlockKind));
  end;
860.
      More abstract methods.
\langle Block object implementation 854 \rangle + \equiv
procedure BlockObj.StartSchemeDemonstration;
  begin end;
procedure BlockObj.FinishSchemeDemonstration;
  begin end;
```

 $\S861$ Mizar Parser ITEM OBJECTS 233

Section 18.2. ITEM OBJECTS

861. 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. 8. UML class diagram for Item object class.

862. Items are a tagged union, tagged by the "kind" of item.

```
 \begin{tabular}{l} $\langle ItemKinds\ (syntax.pas)\ 862 \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 848.

```
863. ⟨Class declaration for Item object 863⟩ ≡
ItemObj = object (StackedObj)
nItemKind: ItemKind;
constructor Init(fItemKind : ItemKind);
procedure Pop; virtual;
destructor Done; virtual;
⟨Method declarations for Item object 867⟩
end
```

This code is used in section 848.

864. 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 864⟩ ≡
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 865 and 868.
This code is used in section 843.
```

234 ITEM OBJECTS Mizar Parser $\S 865$

```
865. Creating an expression in an item is handled with this method. \langle Item object implementation 864 \rangle +\equiv procedure ItemObj.CreateExpression(fExpKind: ExpKind); begin gExpPtr \leftarrow new(ExpressionPtr, Init(fExpKind)); end;
```

8866 Mizar Parser ITEM OBJECTS 235

866. 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 866\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;
```

236 ITEM OBJECTS Mizar Parser $\S 866$

```
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;
```

 $\S866$ Mizar Parser ITEM OBJECTS 237

procedure ProcessTheoremNumber; virtual; procedure ProcessSchemeNumber; virtual; procedure StartJustification; virtual; procedure StartSimpleJustification; virtual; procedure FinishSimpleJustification; virtual;

See also section 1404.

This code is used in sections 867 and 1405.

238 ITEM OBJECTS Mizar Parser §867

```
\langle Method declarations for Item object 867\rangle \equiv
  (Methods overriden by extended Item class 866)
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 863.
```

8868 Mizar Parser ITEM OBJECTS 239

```
\langle \text{ Item object implementation } 864 \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;
```

240 ITEM OBJECTS Mizar Parser §868

```
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;
{\bf procedure}\ {\it ItemObj.FinishGuard};
  begin end;
procedure ItemObj.StartOtherwise;
  begin end;
procedure ItemObj.ProcessEquals;
  begin end;
procedure ItemObj.StartEquals;
  begin end;
{\bf procedure}\ Item Obj. Process Correctness;
  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;
```

 $\S 868$ Mizar Parser ITEM OBJECTS 241

```
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;
```

242 ITEM OBJECTS Mizar Parser §868

```
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;
{\bf procedure}\ {\it ItemObj.FinishSchemeThesis};
  begin end;
procedure ItemObj.StartSchemePremise;
```

8868 Mizar Parser ITEM OBJECTS 243

```
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;
```

244 ITEM OBJECTS Mizar Parser $\S 868$

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; procedure 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;

§869 Mizar Parser EXPRESSIONS 245

Section 18.3. EXPRESSIONS

```
869.
\langle \text{ExpKinds (syntax.pas) } 869 \rangle \equiv
  ExpKind = (exNull, exType, exTerm, exFormula, exResType, exAdjectiveCluster);
This code is used in section 848.
870. \langle Expression class declaration 870 \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 848.
871. Constructor.
\langle \text{Expression constructor 871} \rangle \equiv
constructor ExpressionObj.Init(fExpKind : ExpKind);
  begin nExpKind \leftarrow fExpKind;
  end;
This code is used in section 843.
872. 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 872 \rangle \equiv
procedure ExpressionObj.CreateSubexpression;
  begin gSubexpPtr \leftarrow new(SubexpPtr, Init);
  end;
This code is used in section 843.
```

246 SUBEXPRESSIONS Mizar Parser §873

Section 18.4. SUBEXPRESSIONS

```
873.
\langle Subexpression object class 873\rangle \equiv
  SubexpPtr = \uparrow SubexpObj;
  SubexpObj = \mathbf{object} (StackedObj)
     constructor Init;
     destructor Done; virtual;
     (Empty method declarations for SubexpObj 877)
  end
This code is used in section 848.
874. Constructor. Importantly, constructing a new Subexp object will initialize its Previous field to
point to the global gSubexpPtr object.
\langle Subexpression constructor 874\rangle \equiv
constructor SubexpObj.Init;
  \mathbf{begin}\ \mathit{Previous} \leftarrow \mathit{gSubexpPtr};
  end;
This code is used in section 843.
875. Destructor.
\langle Subexpression destructor 875\rangle \equiv
destructor SubexpObj.Done;
  begin gSubexpPtr \leftarrow SubexpPtr(Previous);
  end;
This code is used in section 843.
```

```
876.
      The remaining methods for subexpression objects are empty.
\langle Methods implemented by subclasses of SubexpObj 876\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;
```

248 SUBEXPRESSIONS Mizar Parser $\S876$

```
{ 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 1567.
This code is used in sections 877 and 1568.
```

§877 Mizar Parser SUBEXPRESSIONS 249

```
877.
       \langle \text{ Empty method declarations for } SubexpObj | 877 \rangle \equiv
  \langle Methods implemented by subclasses of SubexpObj 876\rangle
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 873.

250 SUBEXPRESSIONS Mizar Parser §878

878.

```
\langle Subexpression procedures 878\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;
{\bf procedure}\ {\it 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; procedure SubexpObj.ProcessThe; begin end; **procedure** SubexpObj.StartSelectorTerm; begin end; procedure 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;
```

§878 Mizar Parser

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; **procedure** 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 843.

254 MSCANNER Mizar Parser §879

File 19

MScanner

```
879. We have the MScanner module transform an article (an input file) into a stream of tokens.
\langle \text{ scanner.pas } 748 \rangle + \equiv
         (GNU License 4)
unit mscanner;
         interface
         uses errhan, mobjects, scanner;
                  (Public interface for MScanner 880)
         implementation
         uses mizenv;
                  ⟨Implementation for MScanner 886⟩;
         end.
                           Public types. We have enumerated types for each construction we'll encounter in Mizar.
\langle \text{ Public interface for MScanner 880} \rangle \equiv
type (Token kinds for MScanner 884);
         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);
         ⟨Token type for MScanner 881⟩;
See also sections 882 and 883.
This code is used in section 879.
881. Token type for MScanner.
\langle Token type for MScanner 881\rangle \equiv
          Token = \mathbf{record} \ Kind: \ TokenKind;
                            Nr: integer;
                            Spelling: string;
This code is used in section 880.
```

882 Mizar Parser MSCANNER 255

882. Constants for MScanner $\langle \text{ Public interface for MScanner 880} \rangle + \equiv$ { 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 885⟩; 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; 883. Public facing procedures and global variables. Of particular importance, the global variable gScanner is declared here. $\langle \text{ Public interface for MScanner 880} \rangle + \equiv$ 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;

256 MSCANNER Mizar Parser $\S 884$

884. Token kinds. If I were cleverer, I would have some WEB macros to make this readable.

```
\langle Token kinds for MScanner 884\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 }
```

§884 Mizar Parser MSCANNER 257

```
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}
```

258 MSCANNER Mizar Parser $\S 884$

```
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 }

 $sy_consider$, { #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 } $sy_theorem\,, \quad \{\, \texttt{#236}\, \}$ 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 880.

§885 Mizar Parser MSCANNER 261

885. We have string representation for each of the token kinds, which is useful for debugging purposes.

```
\langle Token names for MScanner 885\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 }
```

262 MSCANNER Mizar Parser $\S 885$

```
11, {2 #50 }
11, {3 #51 }
···, {4 #52 }
···, {5 #53 }
7, {6 #54 }
7, {7 #55 }
7, {8 #56 }
···, {9 #57 }
':', {: #58 }
';', {; #59 }
'now', {< #60 }
'=', { = #61 }
'end', {> #62 }
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 }
1, {L #76 }
1, {M #77 }
1, {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}
```

§885 Mizar Parser

MSCANNER 263

```
``, {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}
'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}
', {#138}
'', {#139}
'if', {#140}
(#141)
(#142)
(#143)
is', {#144}
fare, { #145 }
``, {#146}
'otherwise', \{ #147 \}
7, {#148}
7, {#149}
7, {#150}
7, {#151}
'T152', {#152}
11, { #153 }
```

264 MSCANNER Mizar Parser $\S 885$

```
'`, {#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} 7, {#217}
7, {#217}
77, {#218}
77, {#219}
77, {#220}
77, {#221}
77, {#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 882.

266 §886 **MSCANNER** Mizar Parser

Reading a token. This tokenizes a Mizar article, using the scanner's GetToken method. We can trace this GetToken back to its implementation (§789). This, in turn, depends on the SliceIt method (§772). This method is used to determine the next token in parser.pas's Parse function.

This assumes that StartScanner (§889) has been invoked already, which initializes the CurWord token and other variables.

Also important to observe: the *Kind* of the token is populated here. $\langle \text{Implementation for MScanner 886} \rangle \equiv$ **procedure** ReadToken; **begin** $PrevWord \leftarrow CurWord$; $PrevPos \leftarrow CurPos$; $CurWord \leftarrow AheadWord$; $CurPos \leftarrow AheadPos$; { '_' 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); $gScanner \uparrow. GetToken$; $AheadWord.Kind \leftarrow TokenKind(qScanner\uparrow.fLexem.Kind); AheadWord.Nr \leftarrow qScanner\uparrow.fLexem.Nr;$ $AheadWord.Spelling \leftarrow gScanner \uparrow .fStr; AheadPos \leftarrow gScanner \uparrow .fPos;$ See also sections 887, 888, 889, 890, 891, 892, and 893. This code is used in section 879. 887. 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 886\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 lStructModeMaxArgsSize 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;
```

§888 Mizar Parser MSCANNER 267

```
888.
       We cleanup after using the .prf file.
\langle Implementation for MScanner 886\rangle + \equiv
procedure DisposePrf;
  begin ModeMaxArqs.Done; PredMaxArqs.Done; StructModeMaxArqs.Done;
  end:
889. We construct an MScann object to scan a file.
\langle Implementation for MScanner 886\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:
890. We initialize a scanner for a file.
\langle Implementation for MScanner 886\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 886\rangle + \equiv
procedure CloseSourceFile;
  begin dispose(gScanner, Done);
  end:
892. We can combine the previous functions together to initialize a scanner for a file (an article) and its
dictionary file.
\langle Implementation for MScanner 886\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 (§36).
\langle Implementation for MScanner 886\rangle + \equiv
procedure FinishScanning;
  begin gScanner↑.fIdents.SaveXDct(EnvFileName + ´.idx´); CloseSourceFile; DisposePrf;
  end:
```

268 ABSTRACT SYNTAX Mizar Parser §894

File 20

Abstract Syntax

894. 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.

895. 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 } 895 \rangle \equiv \\ \langle \text{GNU License } 4 \rangle \\ \text{unit } abstract\_syntax; \\ \text{interface uses } errhan, mobjects, syntax; \\ \langle \text{Interface for abstract syntax } 897 \rangle \\ \text{implementation} \\ \langle \text{Implementation of abstract syntax } 896 \rangle \\ \text{end} \ .
```

896. The implementation requires discussing a few "special cases" (variables, qualified segments, adjectives) before getting to the usual syntactic classes (terms, types, formulas).

```
\label{eq:constructor} $\langle \mbox{Implementation of abstract syntax } 896 \rangle \equiv $\langle \mbox{Variable AST constructor } 899 \rangle$ $\langle \mbox{Qualified segment AST constructor } 902 \rangle$ $\langle \mbox{Adjective expression AST constructor } 908 \rangle$ $\langle \mbox{Adjective AST constructor } 912 \rangle$ $\langle \mbox{Negated adjective AST constructor } 910 \rangle$ $\langle \mbox{Implementing term AST } 917 \rangle$ $\langle \mbox{Implementing type AST } 959 \rangle$ $\langle \mbox{Implementing formula AST } 971 \rangle$ $\langle \mbox{Within expression AST implementation } 1011 \rangle$
```

This code is used in section 895.

 $\S 897$ Mizar Parser ABSTRACT SYNTAX 269

897. 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.

898. 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" (§916).

```
\ \langle Variable (abstract syntax tree) \ 898 \rangle \\
VariablePtr = \tau VariableObj;
\ VariableObj = \textbf{object} \text{ (MObject)}
\      nIdent: integer; \{ identifier number \}
\      nVarPos: Position;
\      constructor Init(const \ aPos: Position; \ aIdentNr: integer);
\      end

This code is used in section 897.

899. Constructor.
\( \text{Variable AST constructor 899} \rangle \geq \text{constructor Position; aIdentNr: integer);}
\text{ begin nIdent \( \text{- aIdentNr}; nVarPos \( \text{- aPos;} \)
\text{ end;}

This code is used in section 896.}
```

270 ABSTRACT SYNTAX Mizar Parser $\S 900$

900. 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.

901. 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) 901 \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 903 and 905.
This code is used in section 897.
       Constructor.
\langle Qualified segment AST constructor 902\rangle \equiv
constructor QualifiedSegmentObj.Init(const aPos: Position; aSort: SegmentKind);
  begin nSeqmPos \leftarrow aPos; nSeqmentSort \leftarrow aSort;
  end;
See also sections 904 and 906.
This code is used in section 896.
```

903. 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) 901 ⟩ +≡
ImplicitlyQualifiedSegmentPtr = ↑ImplicitlyQualifiedSegmentObj;
ImplicitlyQualifiedSegmentObj = object (QualifiedSegmentObj)
    nIdentifier: VariablePtr;
    constructor Init(const aPos: Position; aIdentifier: VariablePtr);
    destructor Done; virtual;
end;
```

 $\S904$ Mizar Parser ABSTRACT SYNTAX 271

```
Constructor. The constructors and destructors for implicitly qualified segments are straightforward.
\langle \text{Qualified segment AST constructor } 902 \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:
905. 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) } 901 \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 902\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:
907.
      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) 907\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 897.
```

272 ABSTRACT SYNTAX Mizar Parser $\S908$

```
\langle Adjective expression AST constructor 908\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 896.
909. 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) } 909 \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 897.
910. Constructor.
\langle Negated adjective AST constructor 910\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 896.
911. 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) } 911 \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 897.
```

 $\S912$ Mizar Parser ABSTRACT SYNTAX 273

912. Constructor.

```
 \begin{split} &\langle \operatorname{Adjective AST \ constructor \ 912} \rangle \equiv \\ & \mathbf{constructor} \ \mathit{AdjectiveObj.Init}(\mathbf{const} \ \mathit{aPos: Position}; \ \mathit{aAdjectiveNr: integer}; \ \mathit{aArgs: PList}); \\ & \mathbf{begin} \ \mathit{inheritedInit}(\mathit{aPos}, \mathit{wsAdjective}); \ \mathit{nAdjectiveSymbol} \leftarrow \mathit{aAdjectiveNr}; \ \mathit{nArgs} \leftarrow \mathit{aArgs}; \\ & \mathbf{end}; \\ & \mathbf{destructor} \ \mathit{AdjectiveObj.Done}; \\ & \mathbf{begin} \ \mathit{dispose}(\mathit{nArgs}, \mathit{Done}); \\ & \mathbf{end}; \end{split}
```

This code is used in section 896.

Section 20.1. TERMS (ABSTRACT SYNTAX TREE)

913. 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:

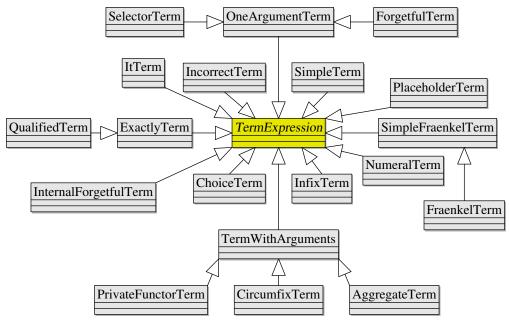


Fig. 9. UML class diagram for abstract syntax tree for terms.

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 913} \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 897}.
```

```
914. 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.
```

```
915.
       Class structure for this syntax tree.
\langle Classes for terms (abstract syntax tree) 915 \rangle \equiv
     { Terms }
  (Simple term (abstract syntax tree) 916);
  (Placeholder term (abstract syntax tree) 918);
   Numeral term (abstract syntax tree) 920);
   Infix term (abstract syntax tree) 922);
   Terms with arguments (abstract syntax tree) 924;
   Circumfix term (abstract syntax tree) 926);
   Private functor term (abstract syntax tree) 928):
   One-argument term (abstract syntax tree) 930;
   Selector term (abstract syntax tree) 932;
   Internal selector term (abstract syntax tree) 934);
   Aggregate term (abstract syntax tree) 936);
   Forgetful functor (abstract syntax tree) 938);
   (Internal forgetful functors (abstract syntax tree) 940);
   Fraenkel terms (abstract syntax tree) 942);
   Exactly term (abstract syntax tree) 948);
   Qualified term (abstract syntax tree) 946);
   Choice term (abstract syntax tree) 950;
  \langle "It" term (abstract syntax tree) 952\rangle;
  (Incorrect term (abstract syntax tree) 954);
This code is used in section 897.
       Simple terms. Mizar describes variables as terms as a Simple Term.
\langle \text{Simple term (abstract syntax tree) } 916 \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 915.
917. Constructors.
\langle \text{Implementing term AST 917} \rangle \equiv
constructor SimpleTermObj.Init(const aPos: Position; aIdentNr: integer);
  begin nTermPos \leftarrow aPos; nTermSort \leftarrow wsSimpleTerm; nIdent \leftarrow aIdentNr;
  end:
See also sections 919, 921, 923, 925, 927, 929, 931, 933, 935, 937, 939, 941, 943, 945, 947, 949, 951, 953, and 955.
This code is used in section 896.
918. 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) } 918 \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 915.
```

919. Constructor.

```
\langle Implementing term AST 917\rangle += constructor PlaceholderTermObj.Init (const aPos: Position; aLocusNr: integer); begin nTermPos \leftarrow aPos; nTermSort \leftarrow wsPlaceholderTerm; nLocusNr \leftarrow aLocusNr; end;
```

920. 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).

```
⟨ Numeral term (abstract syntax tree) 920 ⟩ ≡
NumeralTermPtr = ↑NumeralTermObj;
NumeralTermObj = object (TermExpressionObj)
nValue: integer;
constructor Init(const aPos: Position; aValue: integer);
end
This code is used in section 915.
```

921. Constructor.

```
\langle Implementing term AST 917\rangle +\equiv constructor NumeralTermObj.Init(const aPos: Position; aValue: integer); begin nTermPos \leftarrow aPos; nTermSort \leftarrow wsNumeralTerm; nValue \leftarrow aValue; end:
```

922. 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}) \ 922 \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 915.

923. Constructor.

```
\langle \text{Implementing term AST 917} \rangle +\equiv \\ \textbf{constructor } \textit{InfixTermObj.Init}(\textbf{const } aPos: Position; \\ aFunctorNr: integer; \\ aLeftArgs, aRightArgs: PList); \\ \textbf{begin } nTermPos \leftarrow aPos; nTermSort \leftarrow wsInfixTerm; nFunctorSymbol \leftarrow aFunctorNr; \\ nLeftArgs \leftarrow aLeftArgs; nRightArgs \leftarrow aRightArgs; \\ \textbf{end}; \\ \textbf{destructor } InfixTermObj.Done; \\ \textbf{begin } \textit{dispose}(nLeftArgs, Done); \textit{dispose}(nRightArgs, Done); \\ \textbf{end}; \\ \end{cases}
```

begin dispose(nArgs, Done);

end;

Terms with arguments. This class seems to be used only internally to the abstract_syntax.pas module. Recalling the UML class diagram (§913), 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) 924 $\rangle \equiv$ $TermWithArgumentsPtr = \uparrow TermWithArgumentsObj;$ $TermWithArgumentsObj = \mathbf{object} \ (TermExpressionObj)$ **constructor** *Init*(**const** *aPos*: *Position*; *aKind*: *TermSort*; *aArgs*: *PList*); **destructor** Done: virtual: end This code is used in section 915. 925. Constructor. $\langle \text{Implementing term AST 917} \rangle + \equiv$ 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: 926. 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. $\langle \text{Circumfix term (abstract syntax tree) } 926 \rangle \equiv$ $CircumfixTermPtr = \uparrow CircumfixTermObj;$ $CircumfixTermObj = \mathbf{object} \ (TermWithArgumentsObj)$ nLeftBracketSymbol, nRightBracketSymbol: integer;**constructor** Init(**const** aPos: Position; aLeftBracketNr, aRightBracketNr: integer; aArgs: PList); **destructor** Done; virtual; end This code is used in section 915. 927. Constructor. $\langle \text{Implementing term AST 917} \rangle + \equiv$ **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;

```
928. Private functor terms. We introduce private functor terms in Mizar when we have "defpred
F(\dots) = \dots
\langle Private functor term (abstract syntax tree) 928\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 915.
929.
       Constructor.
\langle \text{Implementing term AST 917} \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 (§913), 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) } 930 \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 915.
931. Constructor.
\langle \text{Implementing term AST 917} \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;
```

932. 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) 932 \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 915.
933. Constructor.
\langle \text{Implementing term AST 917} \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:
934. 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) } 934 \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 915.
935.
      Constructor.
\langle \text{Implementing term AST 917} \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) } 936 \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 915.
```

```
937.
       Constructor.
\langle \text{Implementing term AST 917} \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:
938. 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) 938\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 915.
939.
       Constructor.
\langle \text{Implementing term AST 917} \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;
940. 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) } 940 \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 915.
941. Constructor.
\langle Implementing term AST 917\rangle + \equiv
constructor InternalForgetfulFunctorTermObj.Init(const aPos: Position; aStructSymbol: integer);
  begin nTermPos \leftarrow aPos; nTermSort \leftarrow wsInternalForgetfulFunctorTerm;
  nStructSymbol \leftarrow aStructSymbol;
  end;
```

end;

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) 942 $\rangle \equiv$ $SimpleFraenkelTermPtr = \uparrow SimpleFraenkelTermObj;$ $SimpleFraenkelTermObj = \mathbf{object} \ (TermExpressionObj)$ $nPostqualification: PList; { of segments }$ nSample: TermPtr;**constructor** Init(**const** aPos: Position; aPostqual: PList; aSample: TermPtr); destructor Done; virtual; end:See also section 944. This code is used in section 915. 943. Constructor. $\langle \text{Implementing term AST 917} \rangle + \equiv$ 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; 944. 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) 942 $\rangle + \equiv$ $FraenkelTermPtr = \uparrow FraenkelTermObj;$ $FraenkelTermObj = \mathbf{object} \ (SimpleFraenkelTermObj)$ *nFormula: FormulaPtr*; constructor Init(const aPos: Position; aPostqual: PList; aSample: TermPtr; aFormula: FormulaPtr);**destructor** Done; virtual; end 945. Constructor. $\langle \text{Implementing term AST 917} \rangle + \equiv$ **constructor** FraenkelTermObj.Init(**const** aPos: Position; aPostqual: PList; aSample: TermPtr; aFormula: FormulaPtr); $\textbf{begin} \ nTermPos \leftarrow aPos; \ nTermSort \leftarrow wsFraenkelTerm; \ nPostqualification \leftarrow aPostqual;$ $nSample \leftarrow aSample; nFormula \leftarrow aFormula;$ end; **destructor** FraenkelTermObj.Done; **begin** dispose(nSample, Done); dispose(nPostqualification, Done); dispose(nFormula, Done);

end;

946. Qualified terms. We may wish to explicitly type cast a term (e.g., "term qua newType"), which is what Mizar calls a "qualified term". $\langle \text{Qualified term (abstract syntax tree) } 946 \rangle \equiv$ $Qualified TermPtr = \uparrow Qualified TermObj;$ $QualifiedTermObj = \mathbf{object} (ExactlyTermObj)$ nQualification: TypePtr;**constructor** *Init*(**const** *aPos*: *Position*; *aSubject*: *TermPtr*; *aType*: *TypePtr*); **destructor** Done; virtual; end This code is used in section 915. 947. Constructor. $\langle \text{Implementing term AST 917} \rangle + \equiv$ **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; **948.** Exactly terms. This is the base class for qualified terms. It does not appear to be used anywhere outside the abstract syntax module. $\langle \text{Exactly term (abstract syntax tree) } 948 \rangle \equiv$ $Exactly TermPtr = \uparrow Exactly TermObj;$ $Exactly TermObj = \mathbf{object} \ (TermExpressionObj)$ nSubject: TermPtr;**constructor** *Init*(**const** *aPos*: *Position*; *aSubject*: *TermPtr*); **destructor** Done; virtual; end This code is used in section 915. 949. Constructor. $\langle \text{Implementing term AST 917} \rangle + \equiv$ **constructor** Exactly TermObj. Init(**const** aPos: Position; aSubject: TermPtr); **begin** $nTermPos \leftarrow aPos$; $nTermSort \leftarrow wsExactlyTerm$; $nSubject \leftarrow aSubject$; end; **destructor** ExactlyTermObj.Done; **begin** dispose(nSubject, Done);

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 950. "operates" on soft types instead of arbitrary predicates. \langle Choice term (abstract syntax tree) 950 $\rangle \equiv$ $Choice TermPtr = \uparrow Choice TermObj;$ $ChoiceTermObj = \mathbf{object} \ (TermExpressionObj)$ nChoiceType: TypePtr;**constructor** *Init* (**const** *aPos*: *Position*; *aType*: *TypePtr*); **destructor** Done; virtual; end This code is used in section 915. 951. Constructor. $\langle \text{Implementing term AST 917} \rangle + \equiv$ **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; 952. 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) 952 $\rangle \equiv$ $ItTermPtr = \uparrow ItTermObj;$ $ItTermObj = \mathbf{object} \ (TermExpressionObj)$ **constructor** *Init* (**const** *aPos*: *Position*); end This code is used in section 915. 953. Constructor. $\langle \text{Implementing term AST 917} \rangle + \equiv$ constructor ItTermObj.Init(const aPos: Position); **begin** $nTermPos \leftarrow aPos; nTermSort \leftarrow wsItTerm;$ end; 954. 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. $\langle \text{Incorrect term (abstract syntax tree) } 954 \rangle \equiv$ $IncorrectTermPtr = \uparrow IncorrectTermObj;$ $IncorrectTermObj = \mathbf{object} \ (TermExpressionObj)$ **constructor** *Init* (**const** *aPos*: *Position*); end This code is used in section 915. 955. Constructor. $\langle \text{Implementing term AST 917} \rangle + \equiv$ **constructor** IncorrectTermObj.Init(**const** aPos: Position);

This code is used in section 897.

Section 20.2. TYPES (ABSTRACT SYNTAX TREE)

```
956.
       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 Abstract base class for types 957 \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 897.
958. 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 \dots \langle Adjective_n \rangle \langle Radix\ Type \rangle"
This appears to be used only in definitions.
\langle Classes for type (abstract syntax tree) 958 \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 960, 962, 964, and 966.
```

begin inherited Done;

end;

```
959.
       Constructor.
\langle \text{Implementing type AST 959} \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 961, 963, 965, and 967.
This code is used in section 896.
       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) 958\rangle + \equiv
  StandardTypePtr = \uparrow StandardTypeObj;
  StandardTypeObj = \mathbf{object} (RadixTypeObj)
    nModeSymbol: integer;
    constructor Init(const aPos: Position; aModeSymbol: integer; aArgs: PList);
    destructor Done; virtual;
  end;
961. Constructor.
\langle \text{Implementing type AST 959} \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) 958\rangle + \equiv
  StructTypePtr = \uparrow StructTypeObj;
  StructTypeObj = \mathbf{object} (RadixTypeObj)
    nStructSymbol: integer;
    constructor Init(const aPos: Position; aStructSymbol: integer; aArgs: PList);
    destructor Done; virtual;
  end:
963. Constructor.
\langle \text{Implementing type AST 959} \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;
```

964. Clustered type. The clustered type describes the situation where we accumulate aCluster of adjectives atop aType.

```
\langle Classes for type (abstract syntax tree) 958\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;
965.
       Constructor.
\langle \text{Implementing type AST 959} \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;
966. 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 \text{Classes for type (abstract syntax tree) } 958 \rangle + \equiv
  IncorrectTypePtr = \uparrow IncorrectTypeObj;
  IncorrectTypeObj = \mathbf{object} (TypeExpressionObj)
     constructor Init(const aPos: Position);
  end
967. Constructor.
\langle \text{Implementing type AST 959} \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 968\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 897.
969. 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 .
```

```
970. Right-side of predicative formula.
\langle Classes for formula (abstract syntax tree) 970 \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 972, 974, 976, 978, 980, 982, 984, 986, 988, 990, 992, 994, 996, 998, 1000, 1002, 1004, 1006, and 1008.
This code is used in section 897.
971. Constructor.
\langle \text{Implementing formula AST 971} \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 973, 975, 977, 979, 981, 983, 985, 987, 989, 991, 993, 995, 997, 999, 1001, 1003, 1005, 1007, and 1009.
This code is used in section 896.
972. Predicative formula. A "predicative" formula refers to a formula involving predicates. A predicate
will have a list of terms 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) 970 \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
973.
       Constructor.
\langle \text{Implementing formula AST 971} \rangle + \equiv
constructor PredicativeFormulaObj.Init(const aPos: Position;
                                              aPredNr: integer;
                                              aLeftArgs, aRightArgs: PList);
  begin nFormulaPos \leftarrow aPos; nFormulaSort \leftarrow wsPredicativeFormula; nPredNr \leftarrow aPredNr;
  nLeftArgs \leftarrow aLeftArgs; nRightArgs \leftarrow aRightArgs;
  end:
destructor PredicativeFormulaObj.Done:
```

begin dispose(nLeftArgs, Done); dispose(nRightArgs, Done);

end;

974. 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 " $1 \le i \le len T$ ". This occurs in VECTSP13, for example. $\langle \text{Classes for formula (abstract syntax tree) } 970 \rangle +\equiv$

```
MultiPredicativeFormulaPtr = \uparrow MultiPredicativeFormulaObj;
  MultiPredicativeFormulaObj = \mathbf{object} (FormulaExpressionObj)
    nScraps: PList;
    constructor Init(const aPos: Position; aScraps: PList);
    destructor Done; virtual;
  end
       Constructor.
\langle \text{Implementing formula AST 971} \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) 970 \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
977. Constructor.
\langle \text{Implementing formula AST 971} \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) 970 $\rangle + \equiv$ $Private Predicative Formula Ptr = \uparrow Private Predicative Formula Obj;$ $PrivatePredicativeFormulaObj = \mathbf{object} (FormulaExpressionObj)$ nPredIdNr: integer;nArgs: PList;**constructor** Init(**const** aPos: Position; aPredIdNr: integer; aArgs: PList); **destructor** Done: virtual: end 979. Constructor. $\langle \text{Implementing formula AST 971} \rangle + \equiv$ **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) 970 $\rangle + \equiv$ $QualifyingFormulaPtr = \uparrow QualifyingFormulaObj;$ $QualifyingFormulaObj = \mathbf{object} (FormulaExpressionObj)$ nSubject: TermPtr;nType: TypePtr;constructor Init(const aPos: Position; aSubject: TermPtr; aType: TypePtr); y destructor Done; virtual; end Constructor. $\langle \text{Implementing formula AST 971} \rangle + \equiv$ 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) 970 \rangle + \equiv $NegativeFormulaPtr = \uparrow NegativeFormulaObj;$ $NegativeFormulaObj = \mathbf{object} (FormulaExpressionObj)$ nArg: FormulaPtr;**constructor** *Init* (**const** *aPos*: *Position*; *aArg*: *FormulaPtr*); **destructor** Done; virtual; end 983. Constructor. \langle Implementing formula AST 971 \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; 984. 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) 970 \rangle + \equiv $BinaryFormulaPtr = \uparrow BinaryArgumentsFormula;$ $BinaryArgumentsFormula = \mathbf{object} (FormulaExpressionObj)$ nLeftArg, nRightArg: FormulaPtr; **constructor** *Init* (**const** *aPos*: *Position*; *aLeftArg*, *aRightArg*: *FormulaPtr*); **destructor** Done; virtual; end 985. Constructor. $\langle \text{Implementing formula AST 971} \rangle + \equiv$ **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; 986. Conjunctive formula. A conjunctive formula looks like $\varphi \wedge \psi$ where φ and ψ are logical formulas. \langle Classes for formula (abstract syntax tree) 970 $\rangle + \equiv$ $ConjunctiveFormulaPtr = \uparrow ConjunctiveFormulaObj;$

 $ConjunctiveFormulaObj = \mathbf{object} \ (BinaryArgumentsFormula)$

constructor Init(**const** aPos: Position; aLeftArq, aRightArq: FormulaPtr);

 $\langle \text{Implementing formula AST 971} \rangle + \equiv$

end;

```
987.
       Constructor.
\langle \text{Implementing formula AST 971} \rangle + \equiv
constructor ConjunctiveFormulaObj.Init(const aPos: Position; aLeftArg, aRightArg: FormulaPtr);
  begin inherited Init(aPos, aLeftArq, aRightArq); nFormulaSort \leftarrow wsConjunctiveFormula;
  end:
988. Disjunctive formula. Disjunctive formulas look like \varphi \lor \psi where \varphi and \psi are formulas.
\langle Classes for formula (abstract syntax tree) 970 \rangle + \equiv
  DisjunctiveFormulaPtr = \uparrow DisjunctiveFormulaObj;
  DisjunctiveFormulaObj = \mathbf{object} \ (BinaryArgumentsFormula)
     constructor Init (const aPos: Position; aLeftArg, aRightArg: FormulaPtr);
  end
989.
        Constructor.
\langle \text{Implementing formula AST 971} \rangle + \equiv
constructor DisjunctiveFormulaObj.Init(const aPos: Position;
  aLeftArq, aRightArq: FormulaPtr);
  \textbf{begin} \ \textit{inherited Init} (\textit{aPos}, \textit{aLeftArg}, \textit{aRightArg}); \ \textit{nFormulaSort} \leftarrow \textit{wsDisjunctiveFormula};
  end;
990.
       Conditional formula. Conditional formulas look like \varphi \implies \psi where \varphi and \psi are formulas.
\langle Classes for formula (abstract syntax tree) 970 \rangle + \equiv
  Conditional Formula Ptr = \uparrow Conditional Formula Obj;
  Conditional Formula Obj = \mathbf{object} \ (Binary Arguments Formula)
     constructor Init(const aPos: Position; aLeftArq, aRightArq: FormulaPtr);
  end
991.
       Constructor.
\langle \text{Implementing formula AST 971} \rangle + \equiv
constructor ConditionalFormulaObj.Init(const aPos: Position; aLeftArq, aRightArq: FormulaPtr);
  begin inherited\ Init(aPos, aLeftArg, aRightArg);\ nFormulaSort \leftarrow wsConditionalFormula;
  end:
       Biconditional formula. Biconditional formulas look like \varphi \iff \psi where \varphi and \psi are formulas.
\langle Classes for formula (abstract syntax tree) 970 \rangle + \equiv
  Biconditional Formula Ptr = \uparrow Biconditional Formula Obj;
  Biconditional Formula Obj = \mathbf{object} (Binary Arguments Formula)
     constructor Init (const aPos: Position; aLeftArg, aRightArg: FormulaPtr);
  end
993.
       Constructor.
```

constructor BiconditionalFormulaObj.Init(**const** aPos: Position; aLeftArg, aRightArg: FormulaPtr); **begin** inherited Init(aPos, aLeftArg, aRightArg); nFormulaSort \leftarrow wsBiconditionalFormula;

Mizar Parser

994. 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.

996. 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) 970⟩ +≡
FlexaryDisjunctiveFormulaPtr = ↑FlexaryDisjunctiveFormulaObj;
FlexaryDisjunctiveFormulaObj = object (BinaryArgumentsFormula)
constructor Init(const aPos: Position; aLeftArg, aRightArg: FormulaPtr);
end
```

997. Constructor.

```
\langle Implementing formula AST 971\rangle += constructor FlexaryDisjunctiveFormulaObj.Init(const aPos: Position; aLeftArg, aRightArg: FormulaPtr); begin inheritedInit(aPos, aLeftArg, aRightArg); nFormulaSort \leftarrow wsFlexaryDisjunctiveFormula; end:
```

998. 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".

```
⟨ Classes for formula (abstract syntax tree) 970⟩ +≡
QuantifiedFormulaPtr = ↑QuantifiedFormulaObj;
QuantifiedFormulaObj = object (FormulaExpressionObj)
    nSegment: QualifiedSegmentPtr;
    nScope: FormulaPtr;
    constructor Init(const aPos: Position; aSegment: QualifiedSegmentPtr; aScope: FormulaPtr);
    destructor Done; virtual;
end
```

end

999. Constructor.

```
\langle \text{Implementing formula AST 971} \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:
1000. 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) 970 \rangle + \equiv
  UniversalFormulaPtr = \uparrow UniversalFormulaObj;
  UniversalFormulaObj = \mathbf{object} (QuantifiedFormulaObj)
    constructor Init(const aPos: Position; aSequent: QualifiedSequentPtr; aScope: FormulaPtr);
  end
1001.
        Constructor.
\langle \text{Implementing formula AST 971} \rangle + \equiv
constructor UniversalFormulaObj.Init(const aPos: Position;
                                            aSegment: QualifiedSegmentPtr;
                                            aScope: FormulaPtr);
  begin inherited\ Init(aPos, aSegment, aScope); nFormulaSort \leftarrow wsUniversalFormula;
  end:
1002. 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) 970 \rangle + \equiv
  ExistentialFormulaPtr = \uparrow ExistentialFormulaObj;
  ExistentialFormulaObj = \mathbf{object} (QuantifiedFormulaObj)
    constructor Init(const aPos: Position; aSegment: QualifiedSegmentPtr; aScope: FormulaPtr);
  end
1003.
        Constructor.
\langle \text{Implementing formula AST 971} \rangle + \equiv
constructor ExistentialFormulaObj.Init(const aPos: Position;
                                             aSegment: QualifiedSegmentPtr;
                                             aScope: FormulaPtr);
  begin inherited Init(aPos, aSegment, aScope); nFormulaSort \leftarrow wsExistentialFormula;
  end;
1004.
        Contradiction formula. The canonical contradiction \perp in Mizar is represented by the reserved
keyword "contradiction".
\langle Classes for formula (abstract syntax tree) 970 \rangle + \equiv
  ContradictionFormulaPtr = \uparrow ContradictionFormulaObj;
  ContradictionFormulaObj = \mathbf{object} (FormulaExpressionObj)
    constructor Init (const aPos: Position);
```

end;

```
1005.
         Constructor.
\langle Implementing formula AST 971\rangle + \equiv
constructor ContradictionFormulaObj.Init(const aPos: Position);
  begin nFormulaPos \leftarrow aPos; nFormulaSort \leftarrow wsContradiction;
  end;
         Thesis formula. When we are in the middle of a proof, the goal or obligation left to be proven is
called the "thesis".
\langle Classes for formula (abstract syntax tree) 970 \rangle + \equiv
  ThesisFormulaPtr = \uparrow ThesisFormulaObj;
  ThesisFormulaObj = \mathbf{object} (FormulaExpressionObj)
     constructor Init(const aPos: Position);
  \quad \mathbf{end} \quad
1007.
         Constructor.
\langle \text{Implementing formula AST 971} \rangle + \equiv
constructor ThesisFormulaObj.Init(const aPos: Position);
  begin nFormulaPos \leftarrow aPos; nFormulaSort \leftarrow wsThesis;
  end;
         Incorrect formula. We also have a node in abstract syntax trees for "incorrect" formulas.
\langle Classes for formula (abstract syntax tree) 970 \rangle + \equiv
  IncorrectFormulaPtr = \uparrow IncorrectFormula;
  IncorrectFormula = \mathbf{object} (FormulaExpressionObj)
     constructor Init (const aPos: Position);
  end
1009.
         Constructor.
\langle Implementing formula AST 971\rangle + \equiv
constructor IncorrectFormula.Init(const aPos: Position);
  begin nFormulaPos \leftarrow aPos; nformulaSort \leftarrow wsErrorFormula;
```

Section 20.4. WITHIN EXPRESSIONS (DEFERRED)

1010. The "first identification" process needs to track "within expressions". You should probably come back to this section when you've arrived at the "first identification" unit. \langle Class for Within expression $1010 \rangle \equiv$ $biStackedPtr = \uparrow biStackedObj;$ $biStackedObj = \mathbf{object} (MObject)$ end: $WithinExprPtr = \uparrow WithinExprObj;$ $WithinExprObj = \mathbf{object} \ (MObject)$ nExpKind: ExpKind;nStackArr: array of biStackedPtr; nStackCnt: integer;**constructor** *Init*(*aExpKind* : *ExpKind*); **destructor** Done; virtual; **function** CreateExpressionsVariableLevel: biStackedPtr; virtual; {??} **procedure** Process_Adjective(aAttr: AdjectiveExpressionPtr); virtual; procedure Process_AdjectiveList(aCluster : PList); virtual; **procedure** Process_Variable(var a Var : VariablePtr); virtual; **procedure** Process_ImplicitlyQualifiedVariable(var aSeqm : ImplicitlyQualifiedSeqmentPtr); virtual; **procedure** Process_VariablesSegment(aSegm: QualifiedSegmentPtr); virtual; **procedure** Process_StartVariableSegment; virtual; **procedure** Process_FinishVariableSegment; virtual; **procedure** Process_Type(aTyp: TypePtr); virtual; **procedure** Process_BinaryFormula(aFrm: BinaryFormulaPtr); virtual; **procedure** Process_StartQuantifiedFormula(aFrm: QuantifiedFormulaPtr); virtual; **procedure** Process_QuantifiedFormula(aFrm: QuantifiedFormulaPtr); virtual; **procedure** Process_FinishQuantifiedFormula(aFrm: QuantifiedFormulaPtr); virtual; procedure Process_Formula(aFrm : FormulaPtr); virtual; procedure Process_TermList(aTrmList : PList); virtual; **procedure** Process_SimpleTerm(var aTrm : SimpleTermPtr); virtual; **procedure** Process_StartFraenkelTerm(aTrm: SimpleFraenkelTermPtr); virtual; **procedure** Process_FinishFraenkelTerm(var aTrm : SimpleFraenkelTermPtr); virtual; **procedure** Process_FraenkelTermsScope(var aFrm: FormulaPtr); virtual; **procedure** Process_SimpleFraenkelTerm(var aTrm: SimpleFraenkelTermPtr); virtual; procedure Process_Term(var aTrm : TermPtr); virtual; end; This code is used in section 897. 1011. \langle Within expression AST implementation $1011 \rangle \equiv$

```
constructor WithinExprObj.Init(aExpKind : ExpKind);
  begin setlength(nStackArr, 50); nStackCnt ← 0; nExpKind ← aExpKind;
  end;
See also sections 1012, 1013, 1014, 1015, 1016, 1017, 1018, 1019, 1020, 1021, 1022, 1023, 1024, 1025, 1026, 1027, 1028, and 1029.
This code is used in section 896.

1012. ⟨Within expression AST implementation 1011⟩ +≡
destructor WithinExprObj.Done;
begin inherited Done;
end;
```

```
\langle Within expression AST implementation 1011 \rangle + \equiv
function WithinExprObj.CreateExpressionsVariableLevel: biStackedPtr;
  begin result \leftarrow new(biStackedPtr, Init);
  end;
1014.
        \langle Within expression AST implementation 1011\rangle + \equiv
procedure WithinExprObj.Process_Adjective(aAttr: AdjectiveExpressionPtr);
  begin case aAttr\uparrow.nAdjectiveSort of
  wsAdjective: \mathbf{begin} \ Process\_TermList(AdjectivePtr(aAttr)\uparrow.nArgs); \ \{ nAdjectiveSymbol; \} \}
  wsNegatedAdjective: Process\_Adjective(NegatedAdjectivePtr(aAttr)\uparrow.nArg);
  endcases;
  end:
1015. Within expression AST implementation 1011 \rangle + \equiv
procedure WithinExprObj.Process_AdjectiveList(aCluster: PList);
  var i: integer;
  begin with aCluster \uparrow do
    for i \leftarrow 0 to Count - 1 do Process\_Adjective(Items \uparrow [i]);
  end:
1016. Within expression AST implementation 1011 \rangle + \equiv
procedure WithinExprObj.Process_Variable(var a Var : VariablePtr);
  begin end;
1017. Within expression AST implementation 1011 \rangle + \equiv
procedure Within ExprObj. Process_Implicitly Qualified Variable (var a Segm: Implicitly Qualified Segment Ptr);
  begin Process\_Variable(aSegm \uparrow .nIdentifier);
  end;
        \langle Within expression AST implementation 1011 \rangle + \equiv
procedure WithinExprObj.Process_VariablesSegment(aSegm: QualifiedSegmentPtr);
  var i: integer;
  begin Process_StartVariableSegment;
  case aSegm \uparrow .nSegmentSort of
  ik Impl Qualified Segm:\ Process\_Implicitly Qualified Variable (Implicitly Qualified Segment Ptr(a Segm));
  ikExplQualifiedSegm: with ExplicitlyQualifiedSegmentPtr(aSegm) \uparrow do
       begin for i \leftarrow 0 to nIdentifiers.Count - 1 do Process\_Variable(VariablePtr(nIdentifiers.Items \uparrow [i]));
       Process\_Type(nType);
       end;
  endcases; Process_FinishVariableSegment;
  end:
1019. Within expression AST implementation 1011 \rangle + \equiv
procedure WithinExprObj.Process_StartVariableSegment;
  begin end;
procedure WithinExprObj.Process_FinishVariableSegment;
  begin end;
```

```
\langle Within expression AST implementation 1011 \rangle + \equiv
procedure WithinExprObj.Process_TermList(aTrmList: PList);
  var i: integer;
  begin for i \leftarrow 0 to aTrmList \uparrow . Count - 1 do Process\_Term(TermPtr(aTrmList \uparrow . Items \uparrow [i]));
  end;
1021. Within expression AST implementation 1011 \rangle + \equiv
procedure WithinExprObj.Process_Type(aTyp: TypePtr);
  begin with aTyp\uparrow do
    begin case aTyp\uparrow.nTypeSort of
    wsStandardType: with StandardTypePtr(aTyp) \uparrow do
         begin
                  { nModeSymbol }
         Process\_TermList(nArgs);
         end:
    wsStructureType: with StructTypePtr(aTyp)\uparrow do
                  { nStructSymbol }
         Process\_TermList(nArgs);
         end;
    wsClusteredType: with ClusteredTypePtr(aTyp) \uparrow do
         begin Process_AdjectiveList(nAdjectiveCluster); Process_Type(nType);
         end;
    wsErrorType:;
    endcases;
    end;
  end;
        \langle Within expression AST implementation 1011\rangle + \equiv
procedure WithinExprObj.Process_BinaryFormula(aFrm: BinaryFormulaPtr);
  begin Process\_Formula(aFrm \uparrow .nLeftArq); Process\_Formula(aFrm \uparrow .nRightArq);
  end:
        \langle Within expression AST implementation 1011\rangle + \equiv
procedure WithinExprObj.Process_StartQuantifiedFormula(aFrm: QuantifiedFormulaPtr);
  begin end;
procedure Within ExprObj. Process\_Finish Quantified Formula (aFrm: Quantified Formula Ptr);
  begin end;
1024. Within expression AST implementation 1011 \rangle + \equiv
procedure Within ExprObj. Process_Quantified Formula (aFrm: Quantified Formula Ptr);
  begin Process\_VariablesSegment(aFrm \uparrow .nSegment); Process\_Formula(aFrm \uparrow .nScope);
  end;
```

```
\langle Within expression AST implementation 1011 \rangle + \equiv
procedure WithinExprObj.Process_Formula(aFrm: FormulaPtr);
  var i: integer;
  begin case aFrm \uparrow .nFormulaSort of
  wsNegatedFormula: Process\_Formula(NegativeFormulaPtr(aFrm)\uparrow.nArg);
  wsConjunctiveFormula, wsDisjunctiveFormula, wsConditionalFormula,
         wsBiconditional Formula, wsFlexaryConjunctiveFormula, wsFlexaryDisjunctiveFormula:
         Process\_BinaryFormula(BinaryFormulaPtr(aFrm));
  wsRightSideOfPredicativeFormula: with RightSideOfPredicativeFormulaPtr(aFrm) \uparrow do
      begin
               \{ nPredNr \}
      Process\_TermList(nRightArgs);
      end;
  wsPredicativeFormula: with PredicativeFormulaPtr(aFrm) \uparrow do
      begin Process_TermList(nLeftArgs); { nPredNr }
      Process\_TermList(nRightArgs);
      end;
  wsMultiPredicativeFormula: with MultiPredicativeFormulaPtr(aFrm) \uparrow do
      begin for i \leftarrow 0 to nScraps.Count - 1 do Process\_Formula(nScraps.Items\uparrow[i]);
      end:
  wsPrivatePredicateFormula: with PrivatePredicativeFormulaPtr(aFrm)\uparrow do
      begin
                { nPredIdNr }
      Process\_TermList(nArgs);
      end;
  wsAttributiveFormula: with AttributiveFormulaPtr(aFrm) \uparrow do
      begin Process_Term(nSubject); Process_AdjectiveList(nAdjectives);
      end:
  wsQualifyingFormula: with QualifyingFormulaPtr(aFrm) \uparrow  do
      begin Process_Term(nSubject); Process_Type(nType);
  wsExistentialFormula, wsUniversalFormula: with QuantifiedFormulaPtr(aFrm) \uparrow do
      begin inc(nStackCnt):
      if nStackCnt > length(nStackArr) then setlength(nStackArr, 2 * length(nStackArr));
      nStackArr[nStackCnt] \leftarrow CreateExpressionsVariableLevel;
      Process\_StartQuantifiedFormula(QuantifiedFormulaPtr(aFrm));
      Process\_QuantifiedFormula(QuantifiedFormulaPtr(aFrm));
      Process\_FinishQuantifiedFormula(QuantifiedFormulaPtr(aFrm));
      dispose(nStackArr[nStackCnt], Done); dec(nStackCnt);
      end;
  wsContradiction:;
  wsThesis:;
  wsErrorFormula:;
  endcases;
  end;
```

```
There are a few empty "abstract virtual" methods.
1026.
\langle Within expression AST implementation 1011\rangle + \equiv
procedure WithinExprObj.Process_SimpleTerm(var aTrm : SimpleTermPtr);
  begin end;
procedure \ Within ExprObj. Process\_Start Fraenkel Term (a Trm: Simple Fraenkel Term Ptr);
  begin end;
 \textbf{procedure} \ \ \textit{WithinExprObj.Process\_FinishFraenkelTerm} (\textbf{var} \ a \ \textit{Trm} : SimpleFraenkelTermPtr); 
  begin end;
1027. \langle Within expression AST implementation 1011 \rangle + \equiv
\textbf{procedure} \ \textit{WithinExprObj.Process\_FraenkelTermsScope} (\textbf{var} \ \textit{aFrm} : FormulaPtr);
  begin Process_Formula(aFrm);
  end;
1028. Within expression AST implementation 1011 \rangle + \equiv
procedure WithinExprObj.Process_SimpleFraenkelTerm(var aTrm : SimpleFraenkelTermPtr);
  var i: integer;
  begin with aTrm \uparrow do
     begin for i \leftarrow 0 to nPostqualification \uparrow. Count - 1 do
       Process\_VariablesSegment(QualifiedSegmentPtr(nPostqualification \uparrow. Items \uparrow [i]));
     Process\_Term(nSample);
     end;
  end;
```

```
1029.
       \langle Within expression AST implementation 1011\rangle + \equiv
procedure WithinExprObj.Process_Term(var aTrm: TermPtr);
  begin case aTrm \uparrow .nTermSort of
  wsPlaceholderTerm:  { PlaceholderTermPtr(aTrm)\uparrow.nLocusNr }
  wsSimpleTerm: Process\_SimpleTerm(SimpleTermPtr(aTrm));
  wsNumeralTerm: \{ NumeralTermPtr(aTrm) \uparrow . nValue \}
  wsInfixTerm: with InfixTermPtr(aTrm)\uparrow do
      begin Process\_TermList(nLeftArgs); { nFunctorSymbol }
      Process\_TermList(nRightArgs);
      end:
  wsCircumfixTerm: with CircumfixTermPtr(aTrm) \uparrow do
      begin { nLeftBracketSymbol }
      Process\_TermList(nArgs); \{ nRightBracketSymbol \}
  wsPrivateFunctorTerm: with PrivateFunctorTermPtr(aTrm) \uparrow do
      begin { nFunctorIdent }
      Process\_TermList(nArgs);
      end;
  wsAggregateTerm: with AggregateTermPtr(aTrm) \uparrow do
      begin { nStructSymbol }
      Process\_TermList(nArgs);
      end;
  wsSelectorTerm: with SelectorTermPtr(aTrm)\uparrow do
      begin
              { nSelectorSymbol }
      Process\_Term(nArg);
      end:
  wsInternalSelectorTerm: \{InternalSelectorTermPtr(aTrm)\uparrow.nSelectorSymbol\}
  wsForgetfulFunctorTerm: with ForgetfulFunctorTermPtr(aTrm)\uparrow \mathbf{do}
               { nStructSymbol }
      Process\_Term(nArg);
  wsInternalForgetfulFunctorTerm: \  \{InternalForgetfulFunctorTermPtr(aTrm) \uparrow .nStructSymbol \}
  wsSimpleFraenkelTerm, wsFraenkelTerm: with FraenkelTermPtr(aTrm) \uparrow do
      begin inc(nStackCnt);
      if nStackCnt > length(nStackArr) then setlength(nStackArr, 2 * length(nStackArr));
      nStackArr[nStackCnt] \leftarrow CreateExpressionsVariableLevel;
      Process\_StartFraenkelTerm(SimpleFraenkelTermPtr(aTrm));
      Process\_SimpleFraenkelTerm(SimpleFraenkelTermPtr(aTrm));
      if aTrm \uparrow .nTermSort = wsFraenkelTerm then
         Process\_FraenkelTermsScope(FraenkelTermPtr(aTrm)\uparrow.nFormula);
      Process\_FinishFraenkelTerm(SimpleFraenkelTermPtr(aTrm));
      dispose(nStackArr[nStackCnt], Done); dec(nStackCnt);
  wsQualificationTerm: with QualifiedTermPtr(aTrm)\uparrow do
      begin Process_Term(nSubject); Process_Type(nQualification);
  wsExactlyTerm: Process\_Term(ExactlyTermPtr(aTrm)\uparrow.nSubject);
  wsGlobalChoiceTerm: Process\_Type(ChoiceTermPtr(aTrm)\uparrow.nChoiceType);
  wsItTerm:;
  wsErrorTerm:;
  endcases;
  end;
```

This code is used in section 1030.

Weakly strict Mizar article

1030. 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 1030⟩ ≡
⟨ GNU License 4⟩
unit wsmarticle;
interface
uses mobjects, errhan, mscanner, syntax, abstract_syntax, xml_dict, xml_inout;
⟨ Publicly declared types in wsmarticle.pas 1032⟩
const
⟨ Publicly declared constants in wsmarticle.pas 1035⟩
⟨ Publicly declared functions in wsmarticle.pas 1033⟩
⟨ Publicly declared functions in wsmarticle.pas 1187⟩
implementation
uses mizenv, mconsole, librenv, scanner, xml_parser
mdebug , info end_mdebug;
⟨ Implementation for wsmarticle.pas 1034⟩
end .
```

- 1031. 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.

```
1032.     ⟨Publicly declared types in wsmarticle.pas 1032⟩ ≡
See also sections 1036, 1042, 1044, 1047, 1048, 1050, 1051, 1055, 1057, 1059, 1061, 1064, 1066, 1068, 1070, 1073, 1075, 1077, 1080, 1086, 1089, 1091, 1093, 1095, 1096, 1104, 1106, 1108, 1110, 1126, 1128, 1130, 1132, 1134, 1136, 1138, 1140, 1142, 1144, 1146, 1148, 1150, 1152, 1154, 1156, 1158, 1160, 1162, 1164, 1167, 1169, 1171, 1173, 1175, 1177, 1179, 1181, 1183, 1194, 1277, and 1328.

This code is used in section 1030.

1033.    ⟨Publicly declared functions in wsmarticle.pas 1033⟩ ≡
```

1034. (Implementation for wsmarticle.pas 1034) \equiv

```
See also sections 1037, 1038, 1041, 1043, 1045, 1049, 1052, 1056, 1058, 1060, 1062, 1065, 1067, 1069, 1071, 1074, 1076, 1078, 1081, 1087, 1090, 1092, 1094, 1097, 1098, 1105, 1107, 1109, 1111, 1114, 1116, 1118, 1120, 1122, 1127, 1129, 1131, 1133, 1135, 1137, 1139, 1141, 1143, 1145, 1147, 1149, 1151, 1153, 1155, 1157, 1159, 1161, 1163, 1165, 1168, 1170, 1172, 1174, 1176, 1178, 1180, 1182, 1184, 1185, 1186, 1188, 1193, 1195, 1196, 1197, 1198, 1199, 1200, 1201, 1202, 1203, 1204, 1205, 1206, 1207, 1222, 1223, 1224, 1225, 1238, 1239, 1240, 1241, 1247, 1248, 1249, 1250, 1251, 1252, 1253, 1254, 1255, 1256, 1257, 1258, 1259, 1260, 1275, 1278, 1279, 1280, 1281, 1282, 1283, 1284, 1285, 1286, 1287, 1288, 1289, 1290, 1291, 1292, 1293, 1294, 1295, 1298, 1299, 1300, 1301, 1302, 1303, 1304, 1305, 1306, 1307, 1308, 1309, 1310, 1311, 1312, 1313, 1314, 1315, 1316, 1317, 1318, 1319, 1320, 1321, 1322, 1323, 1324, 1325, 1326, 1327, 1329, 1330, 1331, 1332, 1333, 1334, 1335, 1336, 1337, 1338, 1339, 1340, 1341, 1342, 1343, 1344, 1345, 1346, 1347, 1348, 1349, 1350, 1351, 1352, 1353, 1354, 1355, 1356, 1357, 1358, 1359, 1360, 1361, 1362, 1363, 1364, 1365, 1366, 1367, 1368, 1369, 1370, 1371, and 1372.
```

This code is used in section 1030.

Section 21.1. WEAKLY STRICT TEXT PROPER

1035. Mizar provides a grammar for its syntax in the file

/usr/local/doc/mizar/synta

- 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:

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 (§852). 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.

This code is used in section 1030.

1036. 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 (§852), 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 ($\S\S1123\ et\ seq.$) and Registrations ($\S\S1166\ et\ seq.$).

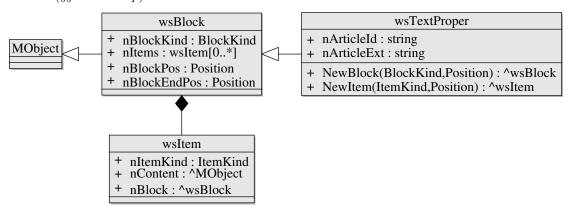


Fig. 10. UML class diagram for wsBlock and related classes.

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 } 1032 \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 1040⟩;
  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;
```

This code is used in section 1036.

1037. Constructor. We initialize using the inherited wsBlock constructor (§1039). 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 1034\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;
```

1038. 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 (§852).

Similarly, when constructing an item which is contained in the block, we need to pass along the item kind (§862).

```
\langle \text{Implementation for wsmarticle.pas } 1034 \rangle + \equiv \\ \text{function } wsTextProper.NewBlock(aBlockKind: BlockKind; const. aPos: Position): } wsBlockPtr; \\ \text{begin } result \leftarrow new(WSBlockPtr, Init(aBlockKind, CurPos)); \\ \text{end}; \\ \text{function } wsTextProper.NewItem(aItemKind: ItemKind; const. aPos: Position): } wsItemPtr; \\ \text{begin } result \leftarrow new(wsItemPtr, Init(aItemKind, CurPos)); \\ \text{end}; \\ \end{cases}
```

1039. Block Constructor. Curiously, the *MObject* constructor ($\S 313$) is not invoked when constructing a wsBlock. We will also need the position ($\S 128$) of the block in the article. The collection of items in the block is initialized to be empty.

```
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;
         Text items. An item requires its "kind" (§862) for its syntactic class.
1040.
\langle \text{Weakly strict Item class } 1040 \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;
```

constructor wsBlock.Init(aBlokKind : BlockKind ; **const** aPos: Position);

```
1041. Constructor
```

```
\langle Implementation for wsmarticle.pas 1034 \rangle + \equiv
constructor wsItem.Init(aItemKind : ItemKind ; const aPos: Position);
  begin nItemKind \leftarrow aItemKind; nItemPos \leftarrow aPos; nItemEndPos \leftarrow aPos; nContent \leftarrow nil;
  nBlock \leftarrow \mathbf{nil};
  end:
destructor wsItem.Done;
  begin if nBlock \neq nil then dispose(nBlock, Done);
  inherited Done;
  end:
1042. Pragmas. Mizar supports pragmas (analogous to conditional compilation).
\langle \text{Publicly declared types in wsmarticle.pas } 1032 \rangle + \equiv
  PragmaPtr = \uparrow PragmaObj;
  PragmaObj = \mathbf{object} \ (MObject)
     nPragmaStr: string;
     constructor Init(aStr : string);
  end;
1043. Constructor.
\langle Implementation for wsmarticle.pas 1034\rangle +=
constructor PragmaObj.Init(aStr : string);
  begin nPragmaStr \leftarrow aStr;
  end:
1044. 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.
\langle \text{Publicly declared types in wsmarticle.pas } 1032 \rangle + \equiv
  LabelPtr = \uparrow LabelObj;
  LabelObj = \mathbf{object} \ (MObject)
     nLabelIdNr: integer;
     nLabelPos: Position:
     constructor Init(aLabelId : integer ; const aPos: Position);
  end:
  PropositionPtr = \uparrow PropositionObj;
  PropositionObj = \mathbf{object} \ (mObject)
     nLab: LabelPtr;
     nSntPos: Position;
     nSentence: FormulaPtr;
     constructor Init(aLab: LabelPtr; aSentence: FormulaPtr; const aSntPos: Position);
     destructor Done; virtual;
  end;
```

1045. Constructor.

```
⟨ Implementation for wsmarticle.pas 1034⟩ +≡ 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;
```

1046. 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,

and scheme references,

$$\langle \mathit{article} \rangle \; ": \mathtt{sch} \; " \; \langle \mathit{number} \rangle$$

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:

```
 \langle Reference \rangle ::= \langle Local\text{-}Reference \rangle \mid \langle Library\text{-}Reference \rangle. \\ \langle Scheme\text{-}Reference \rangle ::= \langle Local\text{-}Scheme\text{-}Reference \rangle \mid \langle Library\text{-}Scheme\text{-}Reference \rangle. \\ \langle Local\text{-}Reference \rangle ::= \langle Label\text{-}Identifier \rangle. \\ \langle Local\text{-}Scheme\text{-}Reference \rangle ::= \langle Scheme\text{-}Identifier \rangle. \\ \langle Library\text{-}Reference \rangle ::= \langle Article\text{-}Name \rangle ":" (\langle Theorem\text{-}Number \rangle) \mid \text{"def" } \langle Definition\text{-}Number \rangle) \\ \{ \text{ "," } (\langle Theorem\text{-}Number \rangle \mid \text{"def" } \langle Definition\text{-}Number \rangle) \} . \\ \langle Library\text{-}Scheme\text{-}Reference \rangle ::= \langle Article\text{-}Name \rangle ":" \text{"sch" } \langle Scheme\text{-}Number \rangle. \\ \end{cases}
```

1047. 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).

```
⟨ Publicly declared types in wsmarticle.pas 1032⟩ +≡
ReferenceKind = (LocalReference, TheoremReference, DefinitionReference);
⟨ Inference kinds (wsmarticle.pas) 1054⟩;
ReferencePtr = ↑ReferenceObj;
ReferenceObj = object (MObject)
    nRefSort: ReferenceKind;
    nRefPos: Position;
end;
```

1048. Local references.

```
⟨ Publicly declared types in wsmarticle.pas 1032 ⟩ +≡
LocalReferencePtr = ↑LocalReferenceObj;
LocalReferenceObj = object (ReferenceObj)
nLabId: integer;
constructor Init(aLabId: integer; const aPos: Position);
end;
```

1049. Constructor. The reference constructors simply populate the appropriate fields in the reference, and the position in the article's text.

```
\langle \text{Implementation for wsmarticle.pas } 1034 \rangle +\equiv 
constructor LocalReferenceObj.Init(aLabId:integer;
const aPos: Position);
begin nRefSort \leftarrow LocalReference; nLabId \leftarrow aLabId; nRefPos \leftarrow aPos
end;
```

1050. Library references. This is the abstract class representing either theorem or definition references from an article.

```
⟨ Publicly declared types in wsmarticle.pas 1032 ⟩ +≡
LibraryReferencePtr = ↑LibraryReferenceObj;
LibraryReferenceObj = object (ReferenceObj)
nArticleNr: integer;
end;
```

1051. 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.

```
⟨ Publicly declared types in wsmarticle.pas 1032⟩ +≡
TheoremReferencePtr = ↑TheoremReferenceObj;
TheoremReferenceObj = object (LibraryReferenceObj)
nTheoNr: integer;
constructor Init(aArticleNr, aTheoNr: integer; const aPos: Position);
end;
DefinitionReferencePtr = ↑DefinitionReferenceObj;
DefinitionReferenceObj = object (LibraryReferenceObj)
nDefNr: integer;
constructor Init(aArticleNr, aDefNr: integer; const aPos: Position);
end;
```

§1052

1052. 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 $1034\rangle + \equiv$ **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** $nRefSort \leftarrow DefinitionReference; nArticleNr \leftarrow aArticleNr; nDefNr \leftarrow aDefNr;$ $nRefPos \leftarrow aPos$ end: 1053. 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. **1054.** The different kinds of inference, since a *Justification* is a tagged union of sorts. $\langle \text{Inference kinds (wsmarticle.pas) } 1054 \rangle \equiv$ InferenceKind = (infError, infStraightforwardJustification, infSchemeJustification, infProof, infProof)infSkippedProof) This code is used in section 1047. 1055. Class structure for justifications. The class hierarchy for justifications reflects the grammar we just discussed. $\langle \text{Publicly declared types in wsmarticle.pas } 1032 \rangle + \equiv$ $JustificationPtr = \uparrow JustificationObj;$ $JustificationObj = \mathbf{object} \ (MObject)$ *nInfSort*: *InferenceKind*; *nInfPos*: Position; **constructor** *Init*(*aInferSort* : *InferenceKind* ; **const** *aPos*: *Position*); end;

```
1056. Constructor. \langle \text{Implementation for wsmarticle.pas } 1034 \rangle +\equiv \text{constructor } JustificationObj.Init(aInferSort: InferenceKind; const } aPos: Position); \\ \text{begin } nInfSort \leftarrow aInferSort; nInfPos \leftarrow aPos; \\ \text{end};
```

```
1057.
        Simple justifications. These are either "by" a list of references, or "from" a scheme.
\langle \text{ Publicly declared types in wsmarticle.pas } 1032 \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;
1058.
        Constructor.
\langle \text{Implementation for wsmarticle.pas } 1034 \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;
1059.
        Straightforward justification.
\langle \text{ Publicly declared types in wsmarticle.pas } 1032 \rangle + \equiv
  StraightforwardJustificationPtr = \uparrow StraightforwardJustificationObj;
  StraightforwardJustificationObj = \mathbf{object} (SimpleJustificationObj)
    nLinked: boolean:
    nLinkPos: Position;
    constructor Init(const aPos: Position; aLinked: boolean; const aLinkPos: Position);
    destructor Done: virtual:
  end;
1060.
        Constructor.
\langle Implementation for wsmarticle.pas 1034\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;
1061.
        Scheme justification.
\langle Publicly declared types in wsmarticle.pas 1032 \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;
    constructor Init(const aPos: Position; aArticleNr, aNr: integer);
    destructor Done; virtual;
  end;
```

1062. Constructor.

```
\langle \text{Implementation for wsmarticle.pas } 1034 \rangle + \equiv \\ \text{constructor } Scheme Justification Obj. Init (const a Pos: Position; a Article Nr, a Nr: integer); \\ \text{begin } inherited Init (inf Scheme Justification, a Pos); n Sch File Nr \leftarrow a Article Nr; n Scheme Id Nr \leftarrow a Nr; \\ n Scheme Inf Pos \leftarrow a Pos; \\ \text{end}; \\ \text{destructor } Scheme Justification Obj. Done; \\ \text{begin } inherited Done; \\ \text{end}; \\ \end{cases}
```

314 SCHEMES Mizar Parser §1063

Section 21.2. SCHEMES

```
1063. 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.
1064. 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 } 1032 \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;
1065. Constructor.
\langle Implementation for wsmarticle.pas 1034 \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;
```

 $\S1066$ Mizar Parser SCHEMES 315

Segment variables for schemes. We need "predicate segments" and "functor segments" for the second-order variable parameters to the scheme. $\langle \text{Publicly declared types in wsmarticle.pas } 1032 \rangle + \equiv$ 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. 1067. $\langle \text{Implementation for wsmarticle.pas } 1034 \rangle + \equiv$ **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. $\langle \text{Publicly declared types in wsmarticle.pas } 1032 \rangle + \equiv$ $SchemePtr = \uparrow SchemeObj;$ $SchemeObj = \mathbf{object} \ (MObject)$ 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;

316 SCHEMES Mizar Parser $\S 1069$

```
1069.
        Constructor.
\langle Implementation for wsmarticle.pas 1034 \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:
1070. 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 } 1032 \rangle + \equiv
  ReservationSegmentPtr = \uparrow ReservationSegmentObj;
  ReservationSegmentObj = \mathbf{object} (MObject)
    nIdentifiers: PList;
    nResType: TypePtr;
    constructor Init(aIdentifiers : PList; aType : TypePtr);
    destructor Done; virtual;
  end;
1071. Constructor.
\langle Implementation for wsmarticle.pas 1034 \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:
```

 $\{1072$ Mizar Parser PRIVATE DEFINITIONS 317

Section 21.3. PRIVATE DEFINITIONS

end;

```
1072. 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).
1073. Private functors.
\langle \text{ Publicly declared types in wsmarticle.pas } 1032 \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;
1074. Constructor.
\langle Implementation for wsmarticle.pas 1034\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:
1075. Private predicates.
\langle \text{Publicly declared types in wsmarticle.pas } 1032 \rangle + \equiv
  PrivatePredicateDefinitionPtr = \uparrow PrivatePredicateDefinitionObj;
  PrivatePredicateDefinitionObj = \mathbf{object} (MObject)
    nPredId: VariablePtr;
    nTypeExpList: PList;
    nSentence: FormulaPtr;
    constructor\ Init(aPredId: VariablePtr;\ aTypeExpList: Plist;\ aSnt: FormulaPtr);
    destructor Done; virtual;
```

318 PRIVATE DEFINITIONS Mizar Parser $\S1076$

```
1076.
        Constructor.
\langle Implementation for wsmarticle.pas 1034\rangle + \equiv
constructor PrivatePredicateDefinitionObj.Init(aPredId : VariablePtr; aTypeExpList : 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;
1077.
        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 } 1032 \rangle + \equiv
  ConstantDefinitionPtr = \uparrow ConstantDefinitionObj;
  ConstantDefinitionObj = \mathbf{object} (MObject)
    nVarId: VariablePtr;
    nTermExpr: TermPtr;
    constructor Init(aVarId : VariablePtr; aTerm : TermPtr);
    destructor Done; virtual;
  end;
1078. Constructor.
\langle Implementation for wsmarticle.pas 1034 \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;
```

 $\{1079$ Mizar Parser CHANGING TYPES 319

Section 21.4. CHANGING TYPES

1079. 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.

1080. Class hierarchy.

```
\langle \text{Publicly declared types in wsmarticle.pas } 1032 \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) 1083⟩
  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;
```

320 CHANGING TYPES Mizar Parser $\S1081$

1081. Constructor. ⟨Implementation for wsmarticle.pas 1034⟩ +≡

```
constructor TypeChangeObj.Init(aKind: TypeChangeSort; aVar: VariablePtr; aTerm: TermPtr); begin nTypeChangeKind \leftarrow aKind; nVar \leftarrow aVar; nTermExpr \leftarrow aTerm; end; destructor TypeChangeObj.Done; begin dispose(nVar, Done); if nTermExpr \neq nil then dispose(nTermExpr, Done); end;
```

 $\langle \, {\rm Constructors} \; {\rm for} \; {\rm example} \; {\rm statements} \; ({\tt wsmarticle.pas}) \; {\tt 1084} \, \rangle$

constructor TypeChangingStatementObj.Init(aTypeChangeList: PList; aTypeExpr: TypePtr; aJustification: SimpleJustificationPtr); **begin** nTypeChangeList \leftarrow aTypeChangeList; nTypeExpr \leftarrow aTypeExpr; nJustification:

begin $nTypeCnangeList \leftarrow aTypeCnangeList; nTypeExpr \leftarrow aTypeExpr nJustification \leftarrow aJustification; end;$

 ${\bf destructor}\ \textit{TypeChangingStatementObj.Done};$

 $\label{eq:begin_dispose} \textbf{begin} \ \textit{dispose}(nTypeChangeList, Done); \ \textit{dispose}(nTypeExpr, Done); \ \textit{dispose}(nJustification, Done); \\ \textbf{end};$

 $\{1082 \quad \text{Mizar Parser} \quad PROOF STEPS \quad 321$

Section 21.5. PROOF STEPS

1082. 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 $\langle formula \rangle$).

1083. 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:

```
Exemplification = "take" Example {"," Example} ";" .
Example = Term-Expression | Variable-Identifier "=" Term-Expression .
\langle \text{Example classes (wsmarticle.pas) } 1083 \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 1080.
1084. Constructor.
\langle \text{Constructors for example statements (wsmarticle.pas)} 1084 \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 1081.
1085. Existential elimination. We continue plugging along with the statements, and existential elimi-
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 } 1032 \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;
```

322 PROOF STEPS Mizar Parser $\{1087$

```
1087. Constructor.
\langle Implementation for wsmarticle.pas 1034\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;
1088.
        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).
       Abstract base class.
1089.
\langle \text{ Publicly declared types in wsmarticle.pas } 1032 \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;
    {\bf constructor}\  \, Init(a Statement Sort: Regular Statement Kind);
    destructor Done; virtual;
  end;
1090. Constructor.
\langle Implementation for wsmarticle.pas 1034\rangle + \equiv
constructor RegularStatementObj.Init(aStatementSort : RegularStatementKind);
  begin nStatementSort \leftarrow aStatementSort;
  end:
destructor RegularStatementObj.Done;
  begin inherited Done;
  end;
1091. 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 } 1032 \rangle + \equiv
  DiffuseStatementPtr = \uparrow DiffuseStatementObj;
  DiffuseStatementObj = \mathbf{object} \ (RegularStatementObj)
    constructor Init(aLab : LabelPtr; aStatementSort : RegularStatementKind);
    destructor Done; virtual;
  end;
```

 $\{1092$ Mizar Parser PROOF STEPS 323

```
1092.
        Constructor.
\langle Implementation for wsmarticle.pas 1034\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;
1093.
        Compact statements. We recall the syntax for a compact statement is:
Compact-Statement = Proposition Justification ";" .
\langle \text{Publicly declared types in wsmarticle.pas } 1032 \rangle + \equiv
  CompactStatementPtr = \uparrow CompactStatementObj;
  CompactStatementObj = \mathbf{object} \ (RegularStatementObj)
    nProp: PropositionPtr;
    nJustification: JustificationPtr;
    constructor Init(aProp : PropositionPtr; aJustification : JustificationPtr);
    destructor Done; virtual;
  end;
1094.
        Constructor.
\langle Implementation for wsmarticle.pas 1034\rangle + \equiv
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:
1095. Iterative equality. Chain of equations, where we keep transforming the right-hand side until we
arrive at the desired outcome.
\langle Publicly declared types in wsmarticle.pas 1032 \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;
1096. \langle \text{Publicly declared types in wsmarticle.pas } 1032 \rangle + \equiv
  IterativeEqualityPtr = \uparrow IterativeEqualityObj;
  IterativeEqualityObj = \mathbf{object} (CompactStatementObj)
    nIterSteps: PList;
    constructor\ Init(aProp: PropositionPtr;\ aJustification: JustificationPtr;\ aIters: PList);
    destructor Done; virtual;
  end;
```

324 PROOF STEPS Mizar Parser $\S1097$

1097. Constructor.

```
\langle Implementation for wsmarticle.pas 1034\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;
1098.
        Constructor.
\langle Implementation for wsmarticle.pas 1034\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;
```

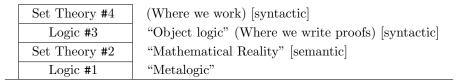
1099. Remaining proof steps? So where are the other proof steps like let or assume? Well, these are handled as "generic text items" and use the *TextItem* class (§1040).

 $\{1100 \quad \text{Mizar Parser} \quad \text{STRUCTURES} \quad 325$

Section 21.6. STRUCTURES

1100. 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. 11. 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.

1101. 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.

326 STRUCTURES Mizar Parser $\S1102$

1102. Aside: finitary metatheory, programming languages, implementing proof assistants.

How does that diagram in Figure 11 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.

1103. 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 .
1104. Field symbol. A "field symbol" refers to the identifier used for a field in a structure, but not its
type.
\langle \text{Publicly declared types in wsmarticle.pas } 1032 \rangle + \equiv
  FieldSymbolPtr = \uparrow FieldSymbolObj;
  FieldSymbolObj = \mathbf{object} \ (MObject)
    nFieldPos: Position;
    nFieldSymbol: integer;
    constructor Init(const aPos: Position; aFieldSymbNr: integer);
  end;
```

1105. Constructor.

```
\langle Implementation for wsmarticle.pas 1034\rangle + \equiv constructor FieldSymbolObj.Init (const aPos: Position; aFieldSymbNr: integer); begin nFieldPos \leftarrow aPos; nFieldSymbol \leftarrow aFieldSymbNr; end;
```

 $\S1106$ Mizar Parser STRUCTURES 327

1106. **Field segment.** A field segment refers to a list of 1 or more field symbols, and the associated type it has. $\langle \text{Publicly declared types in wsmarticle.pas } 1032 \rangle + \equiv$ $FieldSegmentPtr = \uparrow FieldSegmentObj;$ $FieldSegmentObj = \mathbf{object} \ (MObject)$ nFieldSegmPos: Position;nFields: PList;nSpecification: TypePtr;**constructor** *Init* (**const** *aPos*: *Position*; *aFields*: *PList*; *aSpec*: *TypePtr*); **destructor** Done; virtual; end; 1107. Constructor. $\langle Implementation for wsmarticle.pas 1034 \rangle + \equiv$ **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; 1108. Locus. A "locus" refers to a term or type parametrizing a definition. $\langle \text{Publicly declared types in wsmarticle.pas } 1032 \rangle + \equiv$ $LocusPtr = \uparrow LocusObj;$ $LocusObj = \mathbf{object} \ (MObject)$ nVarId: integer;nVarIdPos: Position;**constructor** *Init*(**const** *aPos*: *Position*; *aIdentNr*: *integer*); end; 1109. Constructor. \langle Implementation for wsmarticle.pas 1034 $\rangle + \equiv$ constructor LocusObj.Init(const aPos: Position; aIdentNr: integer); **begin** $nVarId \leftarrow aIdentNr; nVarIdPos \leftarrow aPos;$ end;

328 STRUCTURES Mizar Parser $\S 1110$

1110. 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 } 1032 \rangle + \equiv
  ⟨Pattern objects (wsmarticle.pas) 1113⟩
  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;
1111. Constructor.
\langle Implementation for wsmarticle.pas 1034 \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;
```

 $\{1112 \quad \text{Mizar Parser} \quad 229$

Section 21.7. PATTERNS

```
1112. 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 ")" .
1113. Base class for patterns.
\langle Pattern objects (wsmarticle.pas) 1113 \rangle \equiv
  PatternPtr = \uparrow PatternObj;
  PatternObj = \mathbf{object} \ (mObject)
    nPatternPos: Position;
    nPatternSort: ItemKind;
    constructor Init(const aPos: Position; aSort: ItemKind);
  end;
See also sections 1115, 1117, 1119, and 1121.
This code is used in section 1110.
1114. \langle Implementation for wsmarticle.pas 1034\rangle + \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 ] .
\langle Pattern objects (wsmarticle.pas) 1113 \rangle + \equiv
  ModePatternPtr = \uparrow ModePatternObj;
  ModePatternObj = \mathbf{object} \ (PatternObj)
    nModeSymbol: Integer;
    nArgs: PList;
    constructor Init(const aPos: Position; aSymb: integer; aArgs: PList);
    destructor Done; virtual;
  end;
1116. Constructor.
\langle Implementation for wsmarticle.pas 1034 \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:
```

330 PATTERNS Mizar Parser $\S 1117$

1117. 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 ")" .
\langle Pattern objects (wsmarticle.pas) 1113 \rangle + \equiv
  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;
1118.
        Constructor.
\langle Implementation for wsmarticle.pas 1034 \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:
1119. Predicate patterns. Predicates can have loci on either side of the predicate symbol, without
requiring parentheses (unlike functors).
Predicate-Pattern = [ Loci ] Predicate-Symbol [ Loci ] .
\langle Pattern objects (wsmarticle.pas) 1113 \rangle + \equiv
  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;
1120. Constructor.
\langle \text{Implementation for wsmarticle.pas } 1034 \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;
```

§1121 Mizar Parser PATTERNS 331

1121. 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 ")" .
\langle Pattern objects (wsmarticle.pas) 1113 \rangle + \equiv
  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;
1122.
        Constructor.
\langle Implementation for wsmarticle.pas 1034 \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;
```

332 DEFINITIONS Mizar Parser §1123

Section 21.8. DEFINITIONS

1123. 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 | Attribute-Definition .
```

1124. 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 ) .
```

1125. 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 .
1126. 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 .

 $\{1126 \quad \text{Mizar Parser} \quad \text{DEFINITIONS} \quad 333$

We begin with a base class for definiens. This is extended by SimpleDefiniens and ConditionalDefiniens classes.

```
\langle \text{Publicly declared types in wsmarticle.pas } 1032 \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;
1127.
        Constructor.
\langle Implementation for wsmarticle.pas 1034\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;
        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 } 1032 \rangle + \equiv
  DefExpressionPtr = \uparrow DefExpressionObj;
  DefExpressionObj = \mathbf{object} \ (MObject)
    nExprKind: ExpKind;
    nExpr: PObject;
    constructor Init(aKind : ExpKind; aExpr : PObject);
    destructor Done; virtual;
  end;
1129.
        Constructor.
\langle Implementation for wsmarticle.pas 1034 \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;
```

334 DEFINITIONS Mizar Parser §1130

```
Simple definiens. This is the "default" definiens, i.e., the definiens which are not "by cases".
\langle \text{ Publicly declared types in wsmarticle.pas } 1032 \rangle + \equiv
  SimpleDefiniensPtr = \uparrow SimpleDefiniensObj;
  SimpleDefiniensObj = \mathbf{object} (DefiniensObj)
     nExpression: DefExpressionPtr;
     constructor Init(const aPos: Position; aLab: LabelPtr; aDef: DefExpressionPtr);
     destructor Done; virtual;
  end;
1131.
        Constructor.
\langle \text{Implementation for wsmarticle.pas } 1034 \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;
1132. Definition for particular case. We have "\langle sentence\ or\ term \rangle if \langle quard\ condition \rangle" 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 } 1032 \rangle + \equiv
  PartDefPtr = \uparrow PartDefObj;
  PartDefObj = \mathbf{object} \ (MObject)
     nPartDefiniens: DefExpressionPtr;
     nGuard: FormulaPtr;
     constructor Init(aPartDef : DefExpressionPtr; aGuard : FormulaPtr);
     destructor Done; virtual;
  end;
1133.
        Constructor.
\langle Implementation for wsmarticle.pas 1034\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;
         Conditional definiers. A conditional definiers consists of a finite list of pointers to PartDef
1134.
objects, and a pointer to the default "otherwise" definien.
\langle \text{Publicly declared types in wsmarticle.pas } 1032 \rangle + \equiv
  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;
```

 $\{1135 \quad \text{Mizar Parser} \quad \text{DEFINITIONS} \quad 335$

1135. Constructor.

```
⟨Implementation for wsmarticle.pas 1034⟩ +≡ constructor ConditionalDefiniensObj.Init(const aPos: Position; aLab: LabelPtr; aPartialDefs: PList; aOtherwise: DefExpressionPtr); begin inherited Init(aPos, aLab, ConditionalDefiniens); nConditionalDefiniensList ← aPartialDefs; nOtherwise ← aOtherwise; end; destructor ConditionalDefiniensObj.Done; begin if nOtherwise ≠ nil then dispose(nOtherwise, Done); dispose(nConditionalDefiniensList, Done); inherited Done; end;
```

1136. 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 Publicly declared types in wsmarticle.pas 1032 \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;
1137. Constructor.
\langle Implementation for wsmarticle.pas 1034 \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:
```

336 DEFINITIONS Mizar Parser $\S1138$

```
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.
⟨ Publicly declared types in wsmarticle.pas 1032 ⟩ +≡
  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;
1139. Constructor.
\langle \text{Implementation for wsmarticle.pas } 1034 \rangle + \equiv
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;
1140.
        Standard mode definitions.
\langle \text{Publicly declared types in wsmarticle.pas } 1032 \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;
1141.
        Constructor.
\langle Implementation for wsmarticle.pas 1034 \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:
```

 $\{1142 \quad \text{Mizar Parser} \quad \text{DEFINITIONS} \quad 337$

1142. 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 } 1032 \rangle + \equiv
  AttributeDefinitionPtr = \uparrow AttributeDefinitionObj;
  AttributeDefinitionObj = \mathbf{object} (MObject)
    nDefAttrPos: Position;
    nDefAttrPattern: AttributePatternPtr;
    nRedefinition: boolean;
    nDefiniens: DefiniensPtr;
    constructor Init(const aPos: Position; aRedef: boolean; aPattern: AttributePatternPtr;
       aDef: DefiniensPtr);
    destructor Done; virtual;
  end;
1143. Constructor.
\langle \text{Implementation for wsmarticle.pas } 1034 \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;
1144. 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 } 1032 \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;
```

338 DEFINITIONS Mizar Parser $\S1145$

1145. Constructor.

```
⟨ Implementation for wsmarticle.pas 1034⟩ +≡
constructor PredicateDefinitionObj.Init(const aPos: Position;
    aRedef: boolean; aPattern: PredicatePatternPtr; aDef: DefiniensPtr);
begin nDefPredPos ← aPos; nRedefinition ← aRedef; nDefPredPattern ← aPattern;
nDefiniens ← aDef;
end;
destructor PredicateDefinitionObj.Done;
begin dispose(nDefPredPattern, Done); dispose(nDefiniens, Done);
end:
```

1146. 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) "\(\(new \text{ term}\)\) equals \(\(term \text{ expression}\)\" 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 Publicly declared types in wsmarticle.pas 1032 \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;
```

§1147 DEFINITIONS 339 Mizar Parser

```
1147.
      Constructor.
```

```
\langle Implementation for wsmarticle.pas 1034\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:
1148. 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 } 1032 \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;
1149. Constructor.
\langle Implementation for wsmarticle.pas 1034\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;
```

340 DEFINITIONS Mizar Parser $\S1150$

```
1150. 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 } 1032 \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;
1151. Constructor.
\langle Implementation for wsmarticle.pas 1034\rangle + \equiv
constructor AssumptionObj.Init(const aPos: Position; aSort: AssumptionKind);
  begin nAssumptionPos \leftarrow aPos; nAssumptionSort \leftarrow aSort;
  end:
        Single assumption. When a definition has a single assumption, i.e., a single (usually labeled)
1152.
formula.
\langle \text{Publicly declared types in wsmarticle.pas } 1032 \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;
1153. Constructor.
\langle Implementation for wsmarticle.pas 1034 \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:
```

 $\S1154$ Mizar Parser DEFINITIONS 341

```
1154.
         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 } 1032 \rangle + \equiv
  Collective Assumption Ptr = \uparrow Collective Assumption Obj;
  CollectiveAssumptionObj = \mathbf{object} (AssumptionObj)
    nConditions: PList;
    constructor Init(const aPos: Position; aProps: PList);
    destructor Done; virtual;
  end;
        Constructor.
1155.
\langle \text{Implementation for wsmarticle.pas } 1034 \rangle + \equiv
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;
1156. 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 } 1032 \rangle + \equiv
  ExistentialAssumptionPtr = \uparrow ExistentialAssumptionObj;
  ExistentialAssumptionObj = \mathbf{object} (CollectiveAssumptionObj)
    nQVars: PList;
    constructor Init(const aPos: Position; aQVars, aProps: PList);
    destructor Done; virtual;
  end;
1157.
        Constructor.
\langle Implementation for wsmarticle.pas 1034 \rangle + \equiv
constructor Existential Assumption Obj. Init (const a Pos: Position; a QVars, a Props: 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;
```

342 DEFINITIONS Mizar Parser §1158

```
1158.
        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 } 1032 \rangle + \equiv
  CorrectnessPtr = \uparrow CorrectnessObj;
  CorrectnessObj = \mathbf{object} \ (MObject)
    nCorrCondPos: Position;
    nJustification: JustificationPtr;
    constructor Init(const aPos: Position; aJustification: JustificationPtr);
    destructor Done; virtual;
  end;
1159.
        Constructor.
\langle \text{Implementation for wsmarticle.pas } 1034 \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:
        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 } 1032 \rangle + \equiv
  CorrectnessConditionPtr = \uparrow CorrectnessConditionObj;
  CorrectnessConditionObj = \mathbf{object} (CorrectnessObj)
    nCorrCondSort: CorrectnessKind;
    constructor Init(const aPos: Position; aSort: CorrectnessKind; aJustification: JustificationPtr);
    destructor Done; virtual;
  end;
1161. Constructor.
\langle Implementation for wsmarticle.pas 1034 \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;
```

§1162 Mizar Parser DEFINITIONS 343

1162. Multiple correctness conditions. For, e.g., functors which require proving both "existence" and "uniqueness", we have a *CorrectnessConditions* class. This extends the [singular] *CorrectnessCondition* class.

```
\langle \text{Publicly declared types in wsmarticle.pas } 1032 \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;
1163.
        Constructor.
\langle Implementation for wsmarticle.pas 1034 \rangle + \equiv
constructor CorrectnessConditionsObj.Init(const aPos: Position;
  const aConds: CorrectnessConditionsSet;
       aJustification: JustificationPtr);
  begin inherited Init(aPos, aJustification); nConditions \leftarrow aConds;
destructor CorrectnessConditionsObj.Done;
  begin inherited Done;
  end:
1164.
       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
(§880) 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 } 1032 \rangle + \equiv
  PropertyPtr = \uparrow PropertyObj;
  PropertyObj = \mathbf{object} (MObject)
    nPropertyPos: Position;
    nPropertySort: PropertyKind;
    nJustification: JustificationPtr;
    constructor Init(const aPos: Position; aSort: PropertyKind; aJustification: JustificationPtr);
    destructor Done; virtual;
  end;
```

344 DEFINITIONS Mizar Parser §1165

1165. Constructor.

```
\langle Implementation for wsmarticle.pas 1034\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;
```

§1166 Mizar Parser REGISTRATIONS 345

Section 21.9. REGISTRATIONS

end:

1166. 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 $\langle attributes \rangle$ 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 .
1167. Cluster registration. We have a base class for the three types of cluster registrations.
\langle \text{ Publicly declared types in wsmarticle.pas } 1032 \rangle + \equiv
  ClusterRegistrationKind = (ExistentialRegistration, ConditionalRegistration, FunctorialRegistration);
  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;
```

346 REGISTRATIONS Mizar Parser §1168

```
1168.
        Constructor.
\langle Implementation for wsmarticle.pas 1034\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;
1169.
        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 Publicly declared types in wsmarticle.pas 1032 \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 } 1034 \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);
```

1171. Conditional cluster. For example "empty sets" are always "finite sets". This requires tracking the antecedent ("empty"), and the superclass tracks the consequents ("finite").

```
⟨ Publicly declared types in wsmarticle.pas 1032⟩ +≡
    CClusterPtr = ↑CClusterObj;
    CClusterObj = object (ClusterObj)
        nAntecedent: PList;
    constructor Init(const aPos: Position; aAntec, aCons: PList; aTyp: TypePtr);
    destructor Done; virtual;
end;
```

1172. Constructor.

inherited Done:

end;

```
\langle Implementation for wsmarticle.pas 1034\rangle +\equiv constructor CClusterObj.Init (const aPos: Position; aAntec, aCons: PList; aTyp: TypePtr); begin ClusterObj.Init (aPos, ConditionalRegistration, aCons, aTyp); nAntecedent \leftarrow aAntec; end; destructor CClusterObj.Done; begin dispose (nAntecedent, Done); inheritedDone; end;
```

§1173 Mizar Parser REGISTRATIONS 347

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 Publicly declared types in wsmarticle.pas $1032 \rangle + \equiv$ $FClusterPtr = \uparrow FClusterObj;$ $FClusterObj = \mathbf{object} (ClusterObj)$ nClusterTerm: TermPtr;constructor Init(const aPos: Position; aTrm: TermPtr; aCons: PList; aTyp: TypePtr); **destructor** Done; virtual; end; 1174. Constructor. $\langle \text{Implementation for wsmarticle.pas } 1034 \rangle + \equiv$ 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. $\langle \text{Publicly declared types in wsmarticle.pas } 1032 \rangle + \equiv$ $LociEqualityPtr = \uparrow LociEqualityObj;$ $LociEqualityObj = \mathbf{object} \ (mObject)$ nEqPos: Position;nLeftLocus, nRightLocus: LocusPtr;**constructor** Init(**const** aPos: Position; aLeftLocus, aRightLocus: LocusPtr); **destructor** Done; virtual; end; 1176. Constructor. \langle Implementation for wsmarticle.pas $1034 \rangle + \equiv$ 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:

348 REGISTRATIONS Mizar Parser $\S1177$

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. $\langle \text{Publicly declared types in wsmarticle.pas } 1032 \rangle + \equiv$ $IdentifyRegistrationPtr = \uparrow IdentifyRegistrationObj;$ $IdentifyRegistrationObj = \mathbf{object} \ (mObject)$ nIdentifyPos: Position;nOriginPattern, nNewPattern: PatternPtr; nEqLociList: PList;**constructor** Init(**const** aPos: Position; aNewPatt, aOriqPatt: PatternPtr; aEqList: PList); destructor Done; virtual; end; Constructor. 1178. $\langle \text{Implementation for wsmarticle.pas } 1034 \rangle + \equiv$ **constructor** *IdentifyRegistrationObj.Init*(**const** *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. $\langle \text{Publicly declared types in wsmarticle.pas } 1032 \rangle + \equiv$ $PropertyRegistrationPtr = \uparrow PropertyRegistrationObj;$ $PropertyRegistrationObj = \mathbf{object} \ (mObject)$ nPropertyPos: Position;nPropertySort: PropertyKind; **constructor** *Init* (**const** *aPos*: *Position*; *aKind*: *PropertyKind*); **destructor** Done; virtual; end; 1180. Constructor. $\langle \text{Implementation for wsmarticle.pas } 1034 \rangle + \equiv$ **constructor** PropertyRegistrationObj.Init(**const** aPos: Position; aKind: PropertyKind); **begin** $nPropertyPos \leftarrow aPos; nPropertySort \leftarrow aKind;$ ${\bf destructor}\ \textit{PropertyRegistrationObj.Done};$

begin end;

§1181 Mizar Parser REGISTRATIONS 349

```
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 } 1032 \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;
1182. Constructor.
\langle Implementation for wsmarticle.pas 1034 \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 } 1032 \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;
1184.
        Constructor.
\langle Implementation for wsmarticle.pas 1034 \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;
```

350 HELPER FUNCTIONS Mizar Parser §1185

Section 21.10. HELPER FUNCTIONS

```
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 1034\rangle + \equiv
function CapitalizeName (aName: string): string;
  begin result \leftarrow aName;
  if aName[1] \in [\texttt{`a'...`z'}] then dec(Result[1], ord(\texttt{`a'}) - ord(\texttt{`A'}))
  end;
1186. 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 1034 \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`))
1187.
         We will be populating global variables tracking names of identifiers, modes, and other syntactic
classes.
⟨Global variables publicly declared in wsmarticle.pas 1187⟩ ≡
\mathbf{var}\ IdentifierName, AttributeName, StructureName, ModeName, PredicateName, FunctorName,
         SelectorName, LeftBracketName, RightBracketName, MMLIdentifierName: array of string;
This code is used in section 1030.
1188. We will want to initialize these global variables based on previous passes of the scanner.
\langle \text{Implementation for wsmarticle.pas } 1034 \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 1189);
  ⟨Reset reserved keywords 1191⟩;
    { Identifiers }
  ⟨Initialize identifier names from .idx file 1192⟩;
  end;
```

§1189 Mizar Parser HELPER FUNCTIONS 351

1189. 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 1189⟩ ≡
    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 1190⟩;
end;
close(lDct)
This code is used in section 1188.
```

1190. 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 1189.

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 } 1190 \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 : FunctorName[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 \land \land : AttributeName[lNr] \leftarrow QuoteStrForXML(lString); \\ \quad \text{endcases} \\ \end{aligned}
```

352 HELPER FUNCTIONS Mizar Parser $\S1191$

1191. 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.

\(\text{Reset reserved keywords 1191} \) \equiv \(\text{Reset reserved keywords 1191} \) \(\text{Reset reserved keywords 1191} \)

```
Reset reserved keywords 1191) = AttributeName[StrictSym] \leftarrow `strict'; ModeName[SetSym] \leftarrow `set'; PredicateName[EqualitySym] \leftarrow `='; LeftBracketName[SquareBracket] \leftarrow `['; LeftBracketName[CurlyBracket] \leftarrow `['; LeftBracketName[RoundedBracket] \leftarrow `['; RightBracketName[SquareBracket] \leftarrow `]'; RightBracketName[CurlyBracket] \leftarrow `]'; RightBracketName[RoundedBracket] \leftarrow `]'; This code is used in section 1188.
```

1192. The .idx file provides numbers for the local labels and article names referenced in an article.

```
⟨ Initialize identifier names from .idx file 1192⟩ ≡ 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 1188.
```

1193. 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.

```
\langle \text{ Implementation for wsmarticle.pas } 1034 \rangle + \equiv \\ \text{function } IdentRepr(aIdNr: integer): string; \\ \text{begin } mizassert(2000, aIdNr \leq length(IdentifierName)); \\ \text{if } aIdNr > 0 \text{ then } IdentRepr \leftarrow IdentifierName[aIdNr] \\ \text{else } IdentRepr \leftarrow ``; \\ \text{end}; \\ \end{cases}
```

 $\{1194$ Mizar Parser WRITING WSM XML FILES 353

Section 21.11. WRITING WSM XML FILES

1194.

```
\langle \text{Publicly declared types in wsmarticle.pas } 1032 \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_ReservedVariable(aVar : 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;
```

354 WRITING WSM XML FILES Mizar Parser $\S1195$

1195. 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 \text{Implementation for wsmarticle.pas } 1034 \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 ws TextProper object (\S1036). 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 1034 \rangle + \equiv
procedure OutWSMizFileObj.Out_TextProper(aWSTextProper: WSTextProperPtr);
  var i: integer;
  begin with aWSTextProper↑ 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;
```

§1197 Mizar Parser WRITING WSM XML FILES 355

1197. 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 1034 \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;
       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 1034 \rangle + \equiv
procedure OutWSMizFileObj.Out_TermList(aTrmList: PList);
  begin for i \leftarrow 0 to aTrmList\uparrow.Count - 1 do Out\_Term(aTrmList\uparrow.Items\uparrow[i]);
  end;
```

356 WRITING WSM XML FILES Mizar Parser $\S1199$

1199. 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 1034 \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: \mathbf{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;
```

 $\S1200$ Mizar Parser WRITING WSM XML FILES 357

1200. 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 1034⟩ +≡
procedure OutWSMizFileObj.Out_AdjectiveList(aCluster: PList);
   var i: integer;
   begin Out_XElStart(XMLElemName[elAdjectiveCluster]);
   if aCluster↑.Count = 0 then
        begin Out_XElEndO; 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

§1201

358

```
1201. 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 \text{Implementation for wsmarticle.pas } 1034 \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;
```

359

```
§1201
         Mizar Parser
                                                                       EMITTING XML FOR TYPES
  end;
1202.
       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 1034 \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 1034 \rangle + \equiv
procedure OutWSMizFileObj.Out_ReservedVariable(aVar : VariablePtr);
  begin Out_{-}Variable(aVar);
  end:
1204. 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 1034 \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;
```

360 EMITTING XML FOR TYPES Mizar Parser $\S1205$

1205. Qualified variable segments are either implicitly qualified (hence we use the previous function) or explicitly qualified (which look like " $\langle variable\ list\rangle$ being $\langle type\rangle$ ").

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 \text{Implementation for wsmarticle.pas } 1034 \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;
1206. 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 1034\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

1207. 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 1034 \rangle + \equiv
procedure OutWSMizFileObj.Out_Formula(aFrm:FormulaPtr);
  var i: integer:
  begin case aFrm\uparrow.nFormulaSort of
  wsNegatedFormula: (Emit XML for negated formula (WSM) 1208);
  wsConjunctiveFormula: \( \)Emit XML for conjunction (WSM) 1209 \\;
  wsDisjunctiveFormula: \( \)Emit XML for disjunction (WSM) 1210 \( \);
  wsConditionalFormula: (Emit XML for conditional formula (WSM) 1211);
  wsBiconditionalFormula: \( \) Emit XML for biconditional formula (WSM) 1212 \( \);
  wsFlexaryConjunctiveFormula: \( \) Emit XML for flexary-conjunction (WSM) 1213 \( \);
  wsFlexaryDisjunctiveFormula: \( \) Emit XML for flexary-disjunction (WSM) 1214 \( \);
  wsPredicativeFormula: \( \) Emit XML for predicative formula (WSM) 1215 \( \);
  wsRightSideOfPredicativeFormula: \( \) Emit XML for right-side of predicative formula (WSM) 1216\( \);
  wsMultiPredicativeFormula: (Emit XML for multi-predicative formula (WSM) 1217);
  wsPrivatePredicateFormula: Out_PrivatePredicativeFormula(PrivatePredicativeFormulaPtr(aFrm));
  wsAttributiveFormula: (Emit XML for attributive formula (WSM) 1218);
  wsQualifyingFormula: \( \text{Emit XML for qualifying formula (WSM) 1219} \);
  wsUniversalFormula: (Emit XML for universal formula (WSM) 1220);
  wsExistentialFormula: (Emit XML for existential formula (WSM) 1221);
  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;
```

```
1208.
NegatedFormula = element Negated-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula
\langle Emit XML for negated formula (WSM) 1208\rangle \equiv
  begin Out\_XElStart(FormulaName[wsNegatedFormula]); Out\_PosAsAttrs(aFrm<math>\uparrow.nFormulaPos);
  Out\_XAttrEnd; Out\_Formula(NegativeFormulaPtr(aFrm)\uparrow.nArg);
  Out_XElEnd(FormulaName[wsNegatedFormula]);
  end
This code is used in section 1207.
1209.
ConjunctiveFormula = element Conjunctive-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula,
  Formula
\langle \text{ Emit XML for conjunction (WSM) } 1209 \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 1207.
1210.
DisjunctiveFormula = element Disjunctive-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula,
  Formula
}
\langle \text{ Emit XML for disjunction (WSM) } 1210 \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 1207.
```

```
1211.
ConditionalFormula = element Conditional-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula,
  Formula
}
\langle Emit XML for conditional formula (WSM) 1211\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 1207.
1212.
BiconditionalFormula = element Biconditional-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula.
  Formula
}
\langle \text{Emit XML for biconditional formula (WSM) } 1212 \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 1207.
1213.
FlexaryConjunctiveFormula = element FlexaryConjunctive-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula,
  Formula
}
\langle Emit XML for flexary-conjunction (WSM) 1213\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);
```

This code is used in section 1207.

end

Out_XElEnd(FormulaName[wsFlexaryConjunctiveFormula]);

```
FlexaryDisjunctiveFormula = element FlexaryDisjunctive-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula,
  Formula
}
\langle Emit XML for flexary-disjunction (WSM) 1214\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 1207.
1215.
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 \text{Emit XML for predicative formula (WSM) 1215} \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]);
    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 1207.
```

```
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) 1216\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 1207.
1217.
Multi-Predicative-Formula = element Multi-Predicative-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula*
\langle Emit XML for multi-predicative formula (WSM) 1217\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 1207.
```

```
§1218
         Mizar Parser
                                                                  EMITTING XML FOR FORMULAS
1218.
Attributive-Formula = element Attributive-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term,
  Adjective-Cluster.element
}
\langle Emit XML for attributive formula (WSM) 1218\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 1207.
1219.
Qualifying-Formula = element Qualifying-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term,
  Type,
  Formula
\langle \text{ Emit XML for qualifying formula (WSM) } 1219 \rangle \equiv
  with QualifyingFormulaPtr(aFrm) \uparrow 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 1207.
1220.
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) } 1220 \rangle \equiv
  with QuantifiedFormulaPtr(aFrm)↑ 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]);
```

This code is used in section 1207.

end

```
1221.
```

```
Existential-Quantifier-Formula = element Existential-Quantifier-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Variable-Segment,
  Formula
}

⟨Emit XML for existential formula (WSM) 1221⟩ ≡
  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 1207.

Subsection 21.11.3. Emitting XML for Terms

1222. We begin with simple terms.

```
Term |= element Simple-Term {
   attribute idnr { xsd:integer },
   attribute spelling { text }?,
   attribute col { xsd:integer },
   attribute line { xsd:integer }
}

⟨Implementation for wsmarticle.pas 1034⟩ +≡

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;
```

 $\S1223$ Mizar Parser EMITTING XML FOR TERMS 369

1223. 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 } 1034 \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
     \textbf{else begin } \textit{Out\_XAttrEnd}; \textit{Out\_TermList}(\textit{nArgs}); \textit{Out\_XElEnd}(\textit{TermName}[\textit{wsPrivateFunctorTerm}]); \\
      end;
    end;
  end;
1224.
        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 } 1034 \rangle + \equiv
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;
```

§1225

endcases; end:

```
1225.
       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 } 1034 \rangle + \equiv
procedure OutWSMizFileObj.Out_Term(aTrm : TermPtr);
  var i: integer;
  begin case aTrm\uparrow.nTermSort of
  wsPlaceholderTerm: (Emit XML for placeholder (WSM) 1226);
  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) 1227 \rangle;
  wsCircumfixTerm: \( \)Emit XML for circumfix term (WSM) 1228 \( \);
  wsPrivateFunctorTerm: Out\_PrivateFunctorTerm(PrivateFunctorTermPtr(aTrm));
  wsAggregate Term: (Emit XML for aggregate term (WSM) 1229);
  wsSelectorTerm: (Emit XML for selector term (WSM) 1230);
  wsInternalSelectorTerm: Out\_InternalSelectorTerm(InternalSelectorTermPtr(aTrm));
  wsForgetfulFunctorTerm: (Emit XML for forgetful functor (WSM) 1231);
  wsInternalForgetfulFunctorTerm: (Emit XML for internal forgetful functor (WSM) 1232);
  wsFraenkelTerm: (Emit XML for Fraenkel term (WSM) 1233);
  wsSimpleFraenkelTerm: \( \)Emit XML for simple Fraenkel term (WSM) 1234 \( \);
  wsQualificationTerm: (Emit XML for qualification term (WSM) 1235);
  wsExactlyTerm: (Emit XML for exactly qualification term (WSM) 1236);
  wsGlobalChoiceTerm: \( \)Emit XML for global choice term (WSM) 1237 \( \);
  wsItTerm: Out\_XElWithPos(TermName[wsItTerm], aTrm \uparrow .nTermPos);
  wsErrorTerm: Out_XEl1(TermName[wsErrorTerm]);
```

§1226 Mizar Parser

EMITTING XML FOR TERMS

371

```
1226.
       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) } 1226 \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 1225.
1227.
       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) } 1227 \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 1225.
```

372 EMITTING XML FOR TERMS Mizar Parser $\S1228$

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? } $\langle \text{Emit XML for circumfix term (WSM) } 1228 \rangle \equiv$ 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 1225. 1229. 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 }? } $\langle \text{ Emit XML for aggregate term (WSM) } 1229 \rangle \equiv$ 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 1225.

1230. Terms: selectors.

1228.

Terms: brackets.

 $\{1230$ Mizar Parser EMITTING XML FOR TERMS 373

```
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) } 1230 \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 1225.
1231. 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) 1231\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 1225.
        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) } 1232 \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 1225.
```

§1233

```
1233.
        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) } 1233 \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 1225.
        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) } 1234 \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 1225.
        Terms: qualification.
1235.
Term |= element Qualification-Term {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term,
```

begin Out_XElStart(TermName[wsQualificationTerm]); Out_PosAsAttrs(nTermPos); Out_XAttrEnd; Out_Term(nSubject); Out_Type(nQualification); Out_XElEnd(TermName[wsQualificationTerm]);

This code is used in section 1225.

 $\langle \text{ Emit XML for qualification term (WSM) } 1235 \rangle \equiv$

with $Qualified TermPtr(aTrm) \uparrow do$

Type

end

 $\S1236$ Mizar Parser EMITTING XML FOR TERMS 375

```
1236.
        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) 1236\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 1225.
1237. 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) 1237\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 1225.
Subsection 21.11.4. Emitting XML for text items
1238. Type-lists are needed for text items.
Type-List = element Type-List {
  Type*
\langle \text{Implementation for wsmarticle.pas } 1034 \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]);
  end;
```

376

```
1239. Locus.
Locus = element Locus {
  attribute idnr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer }
}
\langle \text{Implementation for wsmarticle.pas } 1034 \rangle + \equiv
procedure OutWSMizFileObj.Out_Locus(aLocus: LocusPtr);
  begin with aLocus↑ 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;
1240.
Loci = element Loci { Locus* }
\langle Implementation for wsmarticle.pas 1034 \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;
1241. Patterns.
\langle Implementation for wsmarticle.pas 1034 \rangle + \equiv
procedure OutWSMizFileObj.Out_Pattern(aPattern: PatternPtr);
  begin case aPattern \uparrow .nPatternSort of
  itDefPred: (Emit XML for predicate pattern (WSM) 1242);
  itDefFunc: begin case FunctorPatternPtr(aPattern) \underline{\cap-n}.nFunctKind of
    InfixFunctor: (Emit XML for infix functor pattern (WSM) 1243);
    CircumfixFunctor: (Emit XML for bracket functor pattern (WSM) 1244);
    endcases:
    end;
  itDefMode: (Emit XML for mode pattern (WSM) 1245);
itDefAttr: (Emit XML for attribute pattern (WSM) 1246);
  endcases;
  end;
```

```
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) } 1242 \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 1241.
1243.
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) } 1243 \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 1241.
```

378

```
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) 1244\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 1241.
1245.
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) } 1245 \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 1241.
```

1246. 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) 1246} \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 1241.
1247.
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 1034 \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;
```

1248. Emitting XML for definiens.

380

```
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 1034 \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
               case nExprKind of
               exTerm: Out_Term(TermPtr(nExpr));
               exFormula: Out_Formula(FormulaPtr(nExpr));
```

```
endcases;
           end;
      endcases; Out_XElEnd(XMLElemName[elDefiniens]);
  end;
1249.
Proposition = element Proposition {
  Label,
  Formula
}
\langle \text{Implementation for wsmarticle.pas } 1034 \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;
1250.
Local-Reference = element Local-Reference {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  attribute idnr { xsd:integer },
  attribute spelling { text }?
}
\langle \text{Implementation for wsmarticle.pas } 1034 \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;
```

```
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 1034 \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:
```

```
1252.
Link = element Link {
  attribute col { xsd:integer },
  attribute line { xsd:integer }
}
\langle \text{Implementation for wsmarticle.pas } 1034 \rangle + \equiv
procedure OutWSMizFileObj.Out_Link(aInf: JustificationPtr);
  begin with StraightforwardJustificationPtr(aInf) \uparrow do
    if nLinked then
      begin Out_XElStart(XMLElemName[elLink]); Out_PosAsAttrs(nLinkPos); Out_XElEnd0;
      end;
  end;
1253.
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 \text{Implementation for wsmarticle.pas } 1034 \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;
```

```
1254.
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 1034 \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;
```

```
1256.
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 } 1034 \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;
1257.
Variables = element Variables {
  Variable*
\langle Implementation for wsmarticle.pas 1034 \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;
        \langle Implementation for wsmarticle.pas 1034 \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 1034 \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;
```

1260. Emitting XML for item contents. This is used to expedite emitting the XML for a text-item (§1275).

```
\langle Implementation for wsmarticle.pas 1034\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) 1261);
    itTheorem: Out_CompactStatement(CompactStatementPtr(nContent), nBlock);
    itAxiom: begin end;
    it Reservation: \ Out\_ReservationSegment(ReservationSegmentPtr(nContent));
    itSection:;
    itConclusion, itRegularStatement: Out\_RegularStatement(RegularStatementPtr(nContent), nBlock);
    itChoice: (Emit XML for consider contents (WSM) 1262);
    itReconsider: (Emit XML for reconsider contents (WSM) 1263);
    (Emit XML for definition-related items (WSM) 1264);
    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) 1273);
    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 Emit XML for schema (WSM) 1261 \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;
```

This code is used in section 1260.

```
1262.
Item-contents |= Consider-Statement-contents
Consider-Statement-contents =
( Variable-Segment*,
  element Conditions { Proposition },
  Justification
)
\langle Emit XML for consider contents (WSM) 1262\rangle \equiv
  with ChoiceStatementPtr(nContent)↑ 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 1260.
1263.
Item-contents |= Type-Changing-Statement-contents
Type-Changing-Statement-contents =
((element Equality {
    Variable,
    Term
  } | Variable),
 Type)
\langle \text{ Emit XML for reconsider contents (WSM) } 1263 \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);
```

This code is used in section 1260.

Mizar Parser

```
1264. We will need to recall Out_Variable (§1202) fr PrivateFunctorDefinitionObj (§1073).
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) 1264} \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) 1272);
See also sections 1265, 1266, 1267, 1268, 1269, 1270, and 1271.
This code is used in section 1260.
1265.
Item-contents |=
( Variable-Segment*,
  element Conditions { Proposition* } )
\langle Emit XML for definition-related items (WSM) 1264\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;
1266.
Item-contents |= ( Variable?, Term? ) # Exemplification
                                              # percases, correctness-condition
                    Justification
                   Block
                                              # case block
\langle Emit XML for definition-related items (WSM) 1264\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);
```

endcases; end;

```
1267.
Item-contents |=
  element CorrectnessConditions { # sic!
    element Correctness { attribute condition { text } }*,
       Justification }
Justification # Property
\langle Emit XML for definition-related items (WSM) 1264\rangle +\equiv
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);
1268.
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) 1264\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]);
```

end;

```
1269.
Item-contents |=
(element Redefine { }?,
Pattern,
Definiens)
\langle Emit XML for definition-related items (WSM) 1264\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;
1270.
Item-contents |=
(element Redefine { }?,
Pattern,
element Type-Specification { Type }?,
Definiens)
\langle Emit XML for definition-related items (WSM) 1264\rangle +\equiv
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);
```

```
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) 1264\rangle +=
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
```

```
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) } 1272 \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 1264.

1273. 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) 1273\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 1274.
This code is used in section 1260.
```

```
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) 1273\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
```

end;

```
Mizar Parser
1275.
       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 } 1034 \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;
       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);
```

Section 21.12. READING WSM FILES (DEFERRED)

1277. Reading a WSM file amounts to reading an XML file, which means that the XMLInStream class (§688) 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. $\langle \text{Publicly declared types in wsmarticle.pas } 1032 \rangle + \equiv$ $In WSMizFilePtr = \uparrow In WSMizFileObj;$ $InWSMizFileObj = \mathbf{object} (XMLInStreamObj)$ 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;

```
1278.
        Constructor.
\langle Implementation for wsmarticle.pas 1034\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
(\S649), an XMLAttr is just a wrapper around a string nValue.)
\langle Implementation for wsmarticle.pas 1034 \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;
       We can guery for the position of the XML attribute.
\langle Implementation for wsmarticle.pas 1034 \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;
```

```
1281.
        The state of the WSM parser may be described with a handful of lookup tables.
\langle Implementation for wsmarticle.pas 1034 \rangle + \equiv
\mathbf{var}\ ElemLookup\ Table\ ,\ AttrLookup\ Table\ ,\ BlockLook\ Up\ Table\ ,\ ItemLook\ Up\ Table\ ,\ Formula\ KindLookup\ Table\ ,
       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);
  TermKindLookupTable.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.
1282.
\langle Implementation for wsmarticle.pas 1034\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;
```

end;

end:

var *lNr*: *integer*;

function Str2FormulaKind(aStr:string): FormulaSort;

else $Str2FormulaKind \leftarrow wsErrorFormula$;

begin $lNr \leftarrow FormulaKindLookupTable.IndexOfStr(aStr);$ **if** lNr > -1 **then** $Str2FormulaKind \leftarrow FormulaSort(lNr)$

We can recall, from the XML dictionary module ($\S612$), 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 $1034 \rangle + \equiv$ **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; 1284. Like the previous function, this converts the "nr" attribute for a WSM Mizar attribute XML element into a human readable form. $\langle \text{Implementation for wsmarticle.pas } 1034 \rangle + \equiv$ **function** Str2XMLAttrKind(aStr: string): XMLAttrKind; **var** *lNr*: *integer*; **begin** $lNr \leftarrow AttrLookup Table . IndexOfStr(aStr);$ if lNr > -1 then $Str2XMLAttrKind \leftarrow XMLAttrKind(lNr)$ else $Str2XMLAttrKind \leftarrow atUnknown$; end; The "kinds" of different syntactic classes were introduced earlier in wsmarticle.pas, now we want to translate them into human readable form. $\langle \text{Implementation for wsmarticle.pas } 1034 \rangle + \equiv$ **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;

```
1286.
\langle Implementation for wsmarticle.pas 1034 \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
1287.
        Reading a "term list" just iteratively invokes Read_Term (§1301) until all the children have been
read.
\langle \text{Implementation for wsmarticle.pas } 1034 \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:
        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 1034 \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;
1289. Reading a list of adjectives just iterates over the children of an element.
\langle \text{Implementation for wsmarticle.pas } 1034 \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;
```

404 Parsing types Mizar Parser $\S1290$

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 $1034 \rangle + \equiv$ **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 1291. Parsing a variable from XML just requires reading the attributes, since it is an empty-element. \langle Implementation for wsmarticle.pas 1034 $\rangle + \equiv$ **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; Implicitly qualified variables are just wrappers around a variable. \langle Implementation for wsmarticle.pas $1034 \rangle + \equiv$ **function** In WSMizFileObj.Read_ImplicitlyQualifiedSegment: ImplicitlyQualifiedSegmentPtr; var lPos: Position; **begin** $lPos \leftarrow GetAttrPos$; NextElementState;

 $result \leftarrow new(ImplicitlyQualifiedSegmentPtr, Init(lPos, Read_Variable)); NextElementState;$

end:

 $\S1293$ Mizar Parser PARSING FORMULAS 405

```
Recall (§1205) 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 1034\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;
1294. Private predicates are empty elements, so we only need to read their attributes.
\langle Implementation for wsmarticle.pas 1034 \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;
```

406 Parsing formulas Mizar Parser $\S1295$

```
1295.
\langle Implementation for wsmarticle.pas 1034\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 1296⟩;
  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 1297);
  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, lSqm, Read\_Formula)); NextElementState;
  ws \textit{ExistentialFormula: begin } lPos \leftarrow \textit{GetAttrPos}; \ \textit{NextElementState}; \ lSgm \leftarrow \textit{Read\_VariableSegment};
     result \leftarrow new(ExistentialFormulaPtr, Init(lPos, lSqm, 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;
```

§1296 Mizar Parser PARSING FORMULAS 407

```
1296.
        For formulas with binary connectives, we read both arguments.
\langle Parse XML for formula with binary connective 1296\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 1295.
1297.
\langle \text{ Parse XML for predicate-based formula } 1297 \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 1295.

408 PARSING TERMS Mizar Parser $\S1298$

Subsection 21.12.3. Parsing terms

```
1298.
\langle Implementation for wsmarticle.pas 1034\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;
1299.
\langle \text{Implementation for wsmarticle.pas } 1034 \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;
1300.
\langle Implementation for wsmarticle.pas 1034 \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;
```

§1301 Mizar Parser PARSING TERMS 409

```
1301.
```

```
\langle Implementation for wsmarticle.pas 1034 \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 = SegmentKindName[ikImplQualifiedSegm]) \lor (nElName = SegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegme
                           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[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegmentKindName[ikImplQualifiedSegme
                           SegmentKindName[ikExplQualifiedSegm])) do lList \uparrow .Insert(Read\_VariableSegment);
```

410 Parsing terms Mizar Parser $\S1301$

```
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
1302.
\langle \text{Implementation for wsmarticle.pas } 1034 \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:
1303.
\langle Implementation for wsmarticle.pas 1034\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:
1304.
\langle \text{Implementation for wsmarticle.pas } 1034 \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;
```

 $\{1305$ Mizar Parser PARSING TEXT ITEMS 411

```
1305.
\langle Implementation for wsmarticle.pas 1034\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;
1306.
\langle Implementation for wsmarticle.pas 1034 \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;
1307.
\langle Implementation for wsmarticle.pas 1034 \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
         \mathbf{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;
1308.
\langle Implementation for wsmarticle.pas 1034 \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:
```

412 PARSING TEXT ITEMS Mizar Parser §1309

1309.

§1310 Mizar Parser PARSING TEXT ITEMS 413

1310.

```
\langle Implementation for wsmarticle.pas 1034\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;
```

414 PARSING TEXT ITEMS Mizar Parser $\S1311$

```
1311.
\langle Implementation for wsmarticle.pas 1034\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;
1312.
\langle \text{Implementation for wsmarticle.pas } 1034 \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;
1313.
\langle Implementation for wsmarticle.pas 1034 \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;
1314.
\langle Implementation for wsmarticle.pas 1034 \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
       \textbf{else if} \ \textit{nElName} = \textit{ReferenceKindName}[\textit{DefinitionReference}] \ \textbf{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:
```

 $\S1315$ Mizar Parser PARSING TEXT ITEMS 415

```
1315.
\langle Implementation for wsmarticle.pas 1034\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;
1316.
\langle \text{Implementation for wsmarticle.pas } 1034 \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;
1317.
\langle Implementation for wsmarticle.pas 1034 \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:
1318.
\langle Implementation for wsmarticle.pas 1034\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;
1319.
\langle Implementation for wsmarticle.pas 1034\rangle + \equiv
function In WSMizFileObj.Read_SchemeJustification: SchemeJustificationPtr;
  var lInfPos, lPos: Position; lNr, lIdNr: integer;
  begin UnfPos \leftarrow GetAttrPos; Unfty \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;
```

416 PARSING TEXT ITEMS Mizar Parser $\S1320$

```
1320.
\langle Implementation for wsmarticle.pas 1034\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;
1321.
\langle Implementation for wsmarticle.pas 1034 \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;
```

 $\S1322$ Mizar Parser PARSING TEXT ITEMS 417

1322.

```
\langle Implementation for wsmarticle.pas 1034 \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;
```

418 PARSING TEXT ITEMS Mizar Parser $\S1323$

1323.

```
\langle \text{Implementation for wsmarticle.pas } 1034 \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: \mathbf{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);
```

 $\S1323$ Mizar Parser PARSING TEXT ITEMS 419

```
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))));
```

420 PARSING TEXT ITEMS Mizar Parser $\S1323$

```
NextElementState;
                 while (nState = eStart) \land (nElName = XMLElemName[elProposition]) do
                      Collective Assumption Ptr(a Item \uparrow. n Content) \uparrow. n Conditions \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;
```

 $\S1323$ Mizar Parser PARSING TEXT ITEMS 421

```
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; <math>lList \leftarrow Read\_AdjectiveList;
         lCons \leftarrow Read\_AdjectiveList;
          aItem \uparrow .nContent \leftarrow new(CClusterPtr, Init(lPos, lList, lCons, Read\_Type)); NextElementState;
         end
```

422 PARSING TEXT ITEMS Mizar Parser $\S1323$

```
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;
1324.
\langle Implementation for wsmarticle.pas 1034\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;
```

 $\{1325$ Mizar Parser PARSING TEXT ITEMS 423

```
1325.
\langle Implementation for wsmarticle.pas 1034\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;
1326.
\langle Implementation for wsmarticle.pas 1034\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;
1327.
\langle Implementation for wsmarticle.pas 1034 \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;
```

Section 21.13. PRETTYPRINTING WSM FILES (DEFERRED)

1328.

```
\langle \text{ Publicly declared types in wsmarticle.pas } 1032 \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;
```

end;

```
1329.
        Constructor.
\langle Implementation for wsmarticle.pas 1034 \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;
1330.
        \langle Implementation for wsmarticle.pas 1034\rangle + \equiv
procedure WSMizarPrinterObj.Print_Char(aChar: char);
  begin write(nFile, aChar);
  end;
1331. \langle \text{Implementation for wsmarticle.pas } 1034 \rangle + \equiv
procedure WSMizarPrinterObj.Print_NewLine;
  begin writeln(nFile);
  end;
1332. (Implementation for wsmarticle.pas 1034) +\equiv
procedure WSMizarPrinterObj.Print_Number(const aNumber: integer);
  begin write(nFile, aNumber); Print_Char(´□´);
  end:
        The comment is translated from the Polish comment "?? czy na pewno trzeba robic konwersje", so
I may be mistranslating.
\langle \text{Implementation for wsmarticle.pas } 1034 \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;
        \langle Implementation for wsmarticle.pas 1034 \rangle + \equiv
procedure WSMizarPrinterObj.Print_Indent;
  var i: integer:
  begin for i \leftarrow 1 to nIndent do Print\_Char(` \Box `);
  end:
1335. (Implementation for wsmarticle.pas 1034) +\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;
```

```
1336.
         \langle \text{Implementation for wsmarticle.pas } 1034 \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;
1337. (Implementation for wsmarticle.pas 1034) +\equiv
procedure WSMizarPrinterObj.Print_Variable(aVar : VariablePtr);
  begin with aVar\uparrow do
     begin Print_String(IdentRepr(nIdent));
     end:
  end;
1338.
         \langle Implementation for wsmarticle.pas 1034\rangle + \equiv
procedure WSMizarPrinterObj.Print_ImplicitlyQualifiedVariable(aSegm: ImplicitlyQualifiedSegmentPtr);
  begin Print\_Variable(aSegm \uparrow. nIdentifier);
  end:
1339.
         \langle Implementation for wsmarticle.pas 1034 \rangle + \equiv
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;
1340. (Implementation for wsmarticle.pas 1034) +\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 1034 \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^{.Items^{[i]}});
      end:
    Print_String(`)`);
    end;
  end;
        \langle Implementation for wsmarticle.pas 1034 \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;
       \langle Implementation for wsmarticle.pas 1034 \rangle + \equiv
procedure WSMizarPrinterObj.Print_BinaryFormula(aFrm: BinaryFormulaPtr);
  begin Print\_String(``(`); Print\_Formula(aFrm \uparrow .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;
```

```
1344. ⟨Implementation for wsmarticle.pas 1034⟩ +≡
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;
```

```
1345.
        \langle Implementation for wsmarticle.pas 1034 \rangle + \equiv
procedure WSMizarPrinterObj.Print_Formula(aFrm: FormulaPtr);
  var i: Integer; lNeq: boolean; lFrm: FormulaPtr;
  begin case aFrm↑.nFormulaSort of
  wsNegatedFormula: begin Print_String(TokenName[sy_Not]);
    Print\_Formula(NegativeFormulaPtr(aFrm)\uparrow.nArg);
    end:
  wsConjunctiveFormula, wsDisjunctiveFormula, wsConditionalFormula,
         ws Biconditional Formula, ws Flexary Conjunctive Formula, ws Flexary Disjunctive Formula:
         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;
  wsMultiPredicativeFormula: with MultiPredicativeFormulaPtr(aFrm) \uparrow \mathbf{do}
       begin Print\_String(``(`); lFrm \leftarrow nScraps.Items \uparrow [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]; lNeg \leftarrow lFrm \uparrow .nFormulaSort = wsNegatedFormula;
         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;
1346. (Implementation for wsmarticle.pas 1034) +\equiv
procedure WSMizarPrinterObj.Print_SimpleTermTerm(aTrm:SimpleTermPtr);
  begin Print\_String(IdentRepr(SimpleTermPtr(aTrm)\uparrow.nIdent));
  end;
1347. (Implementation for wsmarticle.pas 1034) +\equiv
procedure WSMizarPrinterObj.Print_PrivateFunctorTerm(aTrm: PrivateFunctorTermPtr);
  begin Print_String(IdentRepr(aTrm\finFunctorIdent)); Print_String(\( ( \) ( \) );
  Print\_OpenTermList(aTrm \uparrow .nArgs); Print\_String(`)`);
  end;
```

```
1348.
        \langle Implementation for wsmarticle.pas 1034 \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;
  wsErrorTerm:
  endcases:
  end:
        \langle Implementation for wsmarticle.pas 1034 \rangle + \equiv
1349.
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;
1350. (Implementation for wsmarticle.pas 1034) +\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;
1351. (Implementation for wsmarticle.pas 1034) +\equiv
procedure WSMizarPrinterObj.Print_Proposition(aProp : PropositionPtr);
  begin Print\_Label(aProp\uparrow.nLab); Print\_Formula(aProp\uparrow.nSentence);
  end;
        \langle \text{Implementation for wsmarticle.pas } 1034 \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;
1353. (Implementation for wsmarticle.pas 1034) +\equiv
procedure WSMizarPrinterObj.Print_Linkage;
  begin Print_String(TokenName[sy_Then]);
  end;
```

```
1354.
         \langle Implementation for wsmarticle.pas 1034 \rangle + \equiv
procedure WSMizarPrinterObj.Print_RegularStatement(aRStm: RegularStatementPtr;
          aBlock: wsBlockPtr);
  var i: integer;
  begin case aRStm\uparrow.nStatementSort of
  stDiffuseStatement: \mathbf{begin} \ Print\_Label(DiffuseStatementPtr(aRStm)\uparrow.nLab); \ Print\_Block(aBlock);
     end:
  stCompactStatement: begin
            \textbf{if} \ (\textit{CompactStatementPtr}(\textit{aRStm}) \uparrow. \textit{nJustification} \uparrow. \textit{nInfSort} = \textit{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 1034 \rangle + \equiv
procedure WSMizarPrinterObj.Print_Reference(aRef : LocalReferencePtr);
  begin Print\_String(IdentRepr(aRef \uparrow .nLabId));
  end;
```

```
1356.
        \langle Implementation for wsmarticle.pas 1034 \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;
1357. (Implementation for wsmarticle.pas 1034) +\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 1034 \rangle + \equiv
procedure WSMizarPrinterObj.Print_SchemeNameInJustification(aInf: SchemeJustificationPtr);
  begin Print\_String(IdentRepr(aInf \uparrow .nSchemeIdNr));
  end;
        \langle Implementation for wsmarticle.pas 1034\rangle +=
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 1034 \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:
        \langle Implementation for wsmarticle.pas 1034 \rangle + \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 1034 \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;
1363. (Implementation for wsmarticle.pas 1034) +\equiv
procedure WSMizarPrinterObj.Print_Locus(aLocus: LocusPtr);
  begin with aLocus↑ do
    begin Print\_String(IdentRepr(nVarId));
    end;
  end;
1364. (Implementation for wsmarticle.pas 1034) +\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 1034 \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 1034 \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 1034 \rangle + \equiv
procedure WSMizarPrinterObj.Print_Block(aWSBlock: WSBlockPtr);
  var i, lIndent: integer;
  begin with aWSBlock \uparrow do
    begin lIndent \leftarrow nIndent; Print\_NewLine; 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;
1368. (Implementation for wsmarticle.pas 1034) +\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;
1369. (Implementation for wsmarticle.pas 1034) +\equiv
procedure WSMizarPrinterObj.Print_ReservedType(aResType: TypePtr);
  begin Print_{-}Type(aResType);
  end;
        \langle Implementation for wsmarticle.pas 1034 \rangle + \equiv
procedure WSMizarPrinterObj.Print_SchemeNameInSchemeHead(aSch:SchemePtr);
  begin Print\_String(IdentRepr(aSch \uparrow .nSchemeIdNr));
  end;
```

```
1371.
         \langle \text{Implementation for wsmarticle.pas } 1034 \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 (i > 0) \land (i < nSchemeParams \uparrow. Count - 1) then Print\_String(f, f);
          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;
```

```
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(\(^1\)); 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) \uparrow 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;
```

```
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);
      end;
```

```
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);
```

446

```
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:
  Conditional Registration: 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;
```

448 DETOUR: PRAGMAS Mizar Parser §1373

File 22

Detour: Pragmas

1373. 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;
```

1374. Cancelling a definition or theorem is handled with the "::\$C" pragma, which is administered only by the editors of the MML. For example "::\$CD" will cancel a definition, "CT" will cancel a theorem, and "CS" cancels a scheme.

```
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;
```

 $\S1375$ Mizar Parser DETOUR: PRAGMAS 449

```
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) =  '$P-' then
    begin ProofPragma \leftarrow false;
    end;
  end:
1376. The "::$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) =  $\$+\`\text{then}
    begin SchemePragmaOn.InsertElem(aLine); end;
  if copy(aPrg, 1, 3) =  $S-\tag{then}
    begin SchemePragmaOff.InsertElem(aLine); end;
  end:
        The CompletePragmas 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 b \ge a then
         begin SchemePragmaIntervals.Assign(a,b); f \leftarrow true; break; end;
    if \neg f then SchemePragmaIntervals.Assign(a, aLine);
    end;
```

end;

450 DETOUR: PRAGMAS Mizar Parser $\S1378$

1378. 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

1379. 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.

For what it's worth, this appears to be conventional among compilers in the 1990s to use global variables to control the state of the compiler. For example David Hanson and Christopher Fraser's A Retargetable C Compiler: Design and Implementation (Addison-Wesley, 1995) has quite a few global variables. If we were starting from scratch, it would be more idiomatic to put the state in a Parser class instance, and we could then use this to unit test the parser. This would become conventional more than a decade after Hanson and Fraser's book was published.

[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 } 1379 \rangle \equiv
  ⟨GNU License 4⟩
unit parseraddition;
  interface
  uses syntax, errhan, mobjects, mscanner, abstract_syntax, wsmarticle, xml_inout;
  procedure InitWsMizarArticle;
     Extended block class declaration 1385
     Extended item class declaration 1405
     (Extended subexpression class declaration 1568)
     Extended expression class declaration 1669
  function GetIdentifier: integer;
  function CreateArgs(aBase: integer): PList;
  var (Global variables introduced in parseraddition.pas 1382)
  implementation
  uses mizenv, mconsole, parser, _formats, pragmas
      mdebug, info end_mdebug;
  const MaxSubTermNbr = 64;
  var (Local variables for parser additions 1389)
    (Implementation of parser additions 1380)
  end.
```

```
1380.
        \langle Implementation of parser additions 1380 \rangle \equiv
  (Get the identifier number for current word 1381)
  (Initialize WS Mizar article 1383);
   Extended block implementation 1386
   Extended item implementation 1406 >
   (Extended subexpression implementation 1570)
  (Extended expression implementation 1670)
This code is used in section 1379.
1381. When the current token is an identifier, we should obtain its number. If the current token is not an
identifier, we should return 0. Since the ID numbers for variables (and types and...) are nonzero, returning
0 indicates the current token is not an identifier.
\langle Get the identifier number for current word 1381\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 1380.
1382. 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.
  The gLastWSItem state variable tracks the last statement item.
\langle Global variables introduced in parseraddition.pas 1382\rangle \equiv
gWSTextProper: wsTextProperPtr; { article's text body AST }
gLastWSBlock: WSBlockPtr; { block statement AST }
qLastWSItem: WSItemPtr; { statement AST }
See also sections 1390, 1392, 1407, 1411, 1414, 1420, 1423, 1427, 1436, 1448, 1453, 1467, 1477, 1488, 1496, 1500, 1508, 1516,
```

This code is used in section 1379.

1518, 1520, 1526, 1528, 1547, 1550, 1554, and 1574.

1383. $\langle \text{Initialize WS Mizar article 1383} \rangle \equiv$ procedure InitWsMizarArticle; begin { inintialize global variables which were declared in parseraddition } $gWSTextProper \leftarrow new(wsTextProperPtr, Init(ArticleID, ArticleExt, CurPos)); <math>gLastWSBlock \leftarrow gWSTextProper; gLastWSItem \leftarrow nil; \\ gBlockPtr \leftarrow new(extBlockPtr, Init(blMain));$ { initialize other global variables }

end;
This code is used in section 1380.

 $\{1384$ Mizar Parser EXTENDED BLOCK CLASS 453

Section 23.1. EXTENDED BLOCK CLASS

1384. We extend the Block class (§851) introduced in the syntax.pas unit. Also recall the wsBlock class (§1036) and the wsItem class (§1040).

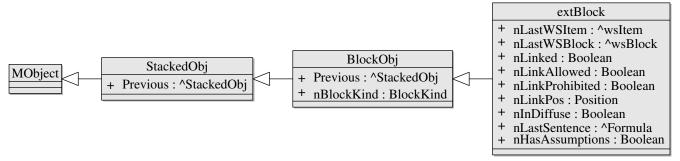


Fig. 12. Class hierarchy for extBlockObj, methods omitted.

```
\langle Extended block class declaration 1385\rangle \equiv
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 1379.

gProofCnt: integer;

1386. Constructor. The constructor for an extended block object invokes the parent class's constructor (§854), initializes the instance variables, then its behaviour depends on whether we are constructing a "main" block or not.

```
\langle Extended block implementation 1386\rangle \equiv
constructor extBlockObj.Init(fBlockKind : BlockKind);
  begin inherited Init (fBlockKind);
  \langle Initialize default values for extBlock instance 1387\rangle;
  if nBlockKind = blMain then \langle Initialize main extBlock instance 1388 \rangle
  else (Initialize "proper text" extBlock instance 1391);
See also sections 1393, 1394, 1395, 1396, 1397, 1398, 1399, 1401, 1402, and 1403.
This code is used in section 1380.
1387. 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 } 1387 \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 1386.
         The "main" block of text needs to load the formats file, and populate the gFormatsColl (§808) 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 } 1388 \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 1386.
1389. \langle Local variables for parser additions 1389\rangle \equiv
Term: array of TermPtr; \{(\S913)\}
See also sections 1408, 1412, 1418, 1421, 1424, 1425, 1428, 1432, 1438, 1440, 1442, 1449, 1451, 1454, 1458, 1464, 1472, 1478,
     1482, 1489, 1497, 1511, 1562, and 1569.
This code is used in section 1379.
         \langle Global variables introduced in parseraddition.pas 1382\rangle + \equiv
```

§1391 Mizar Parser EXTENDED BLOCK CLASS 455

1391. 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 (§449), StackedObj classes has a Previous pointer.)

```
\langle \text{Initialize "proper text" } extBlock \text{ instance } 1391 \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 1386.
```

1392. 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 1382 \rangle += gLastFormula: FormulaPtr;
```

1393. 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 (§855) does nothing. This will be invoked in the KillBlock (§847).

```
\langle Extended block implementation 1386\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;
```

1394. 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:

```
\langle \mathit{Text-Proper} \rangle ::= \langle \mathit{Section} \rangle \ \{ \ \langle \mathit{Section} \rangle \ \} \ .
\langle \mathit{Section} \rangle ::= "begin" \ \{ \ \langle \mathit{Text-Item} \rangle \ \} \ .
```

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 1386\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;
```

1395. This will add a pragma item to the current block. The Parser's *ProcessPragmas* (§1815) invokes this method.

```
⟨ Extended block implementation 1386⟩ +≡ 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;
```

1396. Starting the proper text will just update the nBlockPos field to whatever the current position is.

```
\langle \text{ Extended block implementation } 1386 \rangle + \equiv procedure extBlockObj.StartProperText; begin gWSTextProper \uparrow .nBlockPos \leftarrow CurPos; end;
```

1397. Processing redefinitions sets the global variable gRedefinitions to the result of comparing the current word to the "redefine" keyword.

```
⟨ Extended block implementation 1386⟩ +≡

procedure extBlockObj.ProcessRedefine;

begin gRedefinitions ← CurWord.Kind = sy_Redefine; end;
```

1398. 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 1386\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;
```

§1399 Mizar Parser EXTENDED BLOCK CLASS 457

1399. 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.
```

```
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 Extended block implementation 1386\rangle + \equiv
procedure extBlockObj.StartSchemeDemonstration;
begin inc(gProofCnt);
if \neg ProofPragma then \langle Mark schema proof as "skipped" 1400\rangle;
end;
```

1400. 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" } 1400 \rangle \equiv \\ \textbf{begin } gLastWSItem \leftarrow gWsTextProper \uparrow.NewItem(itConclusion, CurPos);} \\ gLastWSItem \uparrow.nContent \leftarrow new(CompactStatementPtr, Init(thesis\_prop, skipped\_proof\_justification)); \\ gLastWSBlock \uparrow.nItems.Insert(gLastWSItem); \\ \textbf{end}
```

This code is used in section 1399.

1401. Finishing the proof for a scheme should decrement the global "proof depth" counter.

```
⟨ Extended block implementation 1386⟩ +≡ procedure extBlockObj.FinishSchemeDemonstration; begin dec(qProofCnt); end;
```

1402. The factory method for extBlock creating an item will update the global gItemPtr variable (§850).

```
\langle \text{ Extended block implementation } 1386 \rangle + \equiv
procedure extBlockObj.CreateItem(fItemKind : ItemKind);
begin gItemPtr \leftarrow new(extItemPtr, Init(fItemKind)); end;
```

1403. The factory method for extBlock creating a new block will update the gBlockPtr global variable (§850).

```
\langle Extended block implementation 1386\rangle += procedure extBlockObj.CreateBlock(fBlockKind: BlockKind); begin <math>gBlockPtr \leftarrow new(extBlockPtr, Init(fBlockKind)) end;
```

458 EXTENDED ITEM CLASS Mizar Parser $\S1404$

Section 23.2. EXTENDED ITEM CLASS

1404. The class diagram for extended items looks like:

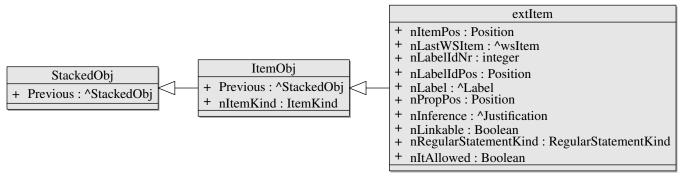


Fig. 13. Class hierarchy for extItemObj. The base MObject class omitted from the hierarchy.

Recall (§1089) 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 866 $\rangle + \equiv$

```
\langle Extended item class declaration 1405 \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 866)
end;
```

This code is used in section 1379.

 $\S1406$ Mizar Parser CONSTRUCTOR 459

Subsection 23.2.1. Constructor

1406. 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 1406\rangle \equiv
constructor extItemObj.Init(fKind : ItemKind);
  begin inherited Init(fKind);
  (Initialize the fields for newly allocated extItem object 1409)
  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 1410)
  endcases;
  if \neg (nItemKind \in [itReservation, itConstantDefinition, itExemplification, itGeneralization,
          itLociDeclaration]) then qLastWSBlock \uparrow .nItems.Insert(qLastWSItem);
  end;
See also sections 1429, 1457, 1459, 1460, 1461, 1462, 1463, 1465, 1466, 1468, 1469, 1470, 1471, 1473, 1474, 1475, 1476, 1479,
     1480, 1481, 1483, 1484, 1485, 1486, 1487, 1490, 1491, 1492, 1493, 1494, 1495, 1498, 1499, 1501, 1502, 1503, 1504, 1505,
     1506, 1507, 1509, 1510, 1512, 1513, 1514, 1515, 1517, 1519, 1521, 1522, 1523, 1524, 1525, 1527, 1529, 1530, 1531, 1532,
     1533,\ 1534,\ 1535,\ 1536,\ 1537,\ 1538,\ 1539,\ 1540,\ 1541,\ 1542,\ 1543,\ 1544,\ 1545,\ 1546,\ 1548,\ 1549,\ 1551,\ 1552,\ 1553,\ 1555,
     1556, 1557, 1558, 1559, 1560, 1561, 1563, 1564, and 1565.
This code is used in section 1380.
```

1407. 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 1382\rangle +\equiv dol_Allowed: Boolean; it_Allowed: Boolean; in_AggrPattern: Boolean; gLastType: TypePtr; gLastTerm: TermPtr; gDefiningWay: HowToDefine; 

1408. \langle Local variables for parser additions 1389\rangle +\equiv gClusterSort: ClusterRegistrationKind; gDefiniens: DefiniensPtr; gPartialDefs: PList; nDefiniensProhibited: boolean; qSpecification: TypePtr;
```

460 CONSTRUCTOR Mizar Parser §1409

```
(Initialize the fields for newly allocated extItem object 1409) \equiv
  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 1406.
1410. 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 | 1410 \rangle \equiv
itGeneralization: ; { let statements }
itExistentialAssumption: ExtBlockPtr(gBlockPtr) \uparrow .nHasAssumptions \leftarrow true;
See also sections 1413, 1415, 1417, 1419, 1422, and 1426.
This code is used in sections 1406 and 1424.
1411. 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 1382\rangle + \equiv
gDefKind: ItemKind;
1412. \langle \text{Local variables for parser additions } 1389 \rangle + \equiv
qExpandable: boolean;
gPropertySort: PropertyKind;
1413. (Initialize extended item by ItemKind 1410) +\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;
```

§1414 Mizar Parser CONSTRUCTOR 461

1414. Reconsider initialization. We need to allocate a new (empty) list for the list of terms being reconsidered.

```
\langle Global variables introduced in parseraddition.pas 1382 \rangle += gReconsiderList\colon PList;
```

- **1415.** (Initialize extended item by $ItemKind\ 1410$) $+\equiv itReconsider: gReconsiderList \leftarrow new(PList, Init(0));$
- 1416. 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.

```
1417. \langle \text{Initialize extended item by } \textit{ItemKind } 1410 \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:;
```

1418. 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.

```
\langle Local variables for parser additions 1389\rangle +\equiv gPremises: PList;
```

gStructPrefixes: PList;

- **1419.** $\langle \text{Initialize extended item by } ItemKind 1410 \rangle + \equiv itAssumption:$ **begin if** $AheadWord.Kind <math>\neq sy_That$ **then** $nLinkable \leftarrow true;$ $gPremises \leftarrow \textbf{nil};$ **end**;
- 1420. 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.

462 CONSTRUCTOR Mizar Parser §1422

```
1422.
        \langle \text{Initialize extended item by } ItemKind | 1410 \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;
1423. 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 1382\rangle + \equiv
qRedefinitions: boolean;
1424. \langle Local variables for parser additions 1389 \rangle + \equiv
gCorrCondSort: CorrectnessKind;
\langle \text{Initialize extended item by } ItemKind | 1410 \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);
        The last statement needing attention will be the scheme block. Note that gLocalScheme is not used
anywhere.
\langle \text{Local variables for parser additions } 1389 \rangle + \equiv
gLocalScheme: boolean;
gSchemePos: Position;
1426. (Initialize extended item by ItemKind 1410) +\equiv
itDefinition, itSchemeHead, itReservation, itPrivFuncDefinition, itPrivPredDefinition, itConstantDefinition,
       itExemplification: ;
itCaseBlock:;
itSchemeBlock: begin gLocalScheme \leftarrow CurWord.Kind \neq sy\_Scheme; gSchemePos \leftarrow CurPos; end;
1427. Popping an extended item.
\langle Global variables introduced in parseraddition.pas 1382\rangle + \equiv
qSchemeParams: PList;
```

§1428 Mizar Parser CONSTRUCTOR 463

```
 \begin{array}{ll} \textbf{1428.} & \langle \operatorname{Local \ variables \ for \ parser \ additions \ 1389} \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

1429. 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 1406⟩ +≡
procedure extItemObj.Pop;
var k: integer;
begin gLastWSItem↑.nItemEndPos ← PrevPos; ⟨Check for errors with definition items 1433⟩
⟨Update content of nLastWSItem based on type of item popped 1430⟩;
⟨Check the popped item's linkages are valid 1456⟩;
if gDefiningWay ≠ dfEmpty then
begin if gDefiniens↑.nDefSort = ConditionalDefiniens then
include(gCorrectnessConditions, syConsistency);
if gRedefinitions then include(gCorrectnessConditions, syCompatibility);
end;
inherited Pop; {(§864)}
end;
```

464 POPPING Mizar Parser $\S1430$

```
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 } 1430 \rangle \equiv
    case nItemKind of
    itTheorem: nLastWSItem \uparrow. nContent \leftarrow new(CompactStatementPtr, Init(new(PropositionPtr,
                  Init(nLabel, gLastFormula, nPropPos)), nInference));
    (Pop a proof step 1434)
    itConclusion, itRegularStatement: (Pop a conclusion or regular statement 1441)
    itGeneralization, itLociDeclaration: (Pop a "let" statement 1443)
    (Pop a definition item 1444)
    itPredSynonym, itPredAntonym, itFuncNotation, itModeNotation, itAttrSynonym, itAttrAntonym:
                  nLastWSItem \uparrow. nContent \leftarrow new(NotationDeclarationPtr, Init(gNewPatternPos, nItemKind,
                  gNewPattern, gPattern));
    (Pop a registration item 1452)
    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 1431⟩
    endcases
This code is used in section 1429.
1431. \langle \text{Pop skips remaining cases 1431} \rangle \equiv
itPrivFuncDefinition, itPrivPredDefinition, itPragma, itDefinition, itSchemeBlock, itReservation,
             itExemplification, itCaseBlock:;
This code is used in section 1430.
                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 } 1389 \rangle + \equiv
gMeansPos: Position;
               \langle Check for errors with definition items 1433 \rangle \equiv
    case nItemKind of
    itDefPred, itDefFunc, itDefMode, itDefAttr: begin if gDefiningWay \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 1429.
```

 $\S1434$ Mizar Parser POPPING 465

1434. 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 1434 \rangle \equiv itPerCases: nLastWSItem \uparrow. nContent \leftarrow nInference; See also sections 1435, 1437, and 1439.
This code is used in section 1430.
```

1435. 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 } 1434 \rangle + \equiv itReconsider: nLastWSItem \uparrow .nContent \leftarrow new(TypeChangingStatementPtr, Init(gReconsiderList, gLastType, SimpleJustificationPtr(nInference)));
```

1436. 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 Qualified\text{-}Segment \rangle ::= \langle Variables \rangle \langle Qualification \rangle
\langle Variables \rangle ::= \langle Variable \rangle \{ "," \langle Variable \rangle \}
\langle Qualification \rangle ::= ("being" | "be") \langle Type\text{-}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 1382\rangle +\equiv gQualifiedSegmentList: PList;
```

```
1437. \langle \text{Pop a proof step } 1434 \rangle + \equiv
```

```
itChoice: \mathbf{begin} \ nLastWSItem \uparrow. nContent \leftarrow new(ChoiceStatementPtr, Init(gQualifiedSegmentList, gPremises, SimpleJustificationPtr(nInference))); gPremises \leftarrow \mathbf{nil}; end:
```

```
itExistentialAssumption: begin nLastWSItem \uparrow .nContent \leftarrow new(ExistentialAssumptionPtr, Init(nItemPos, gQualifiedSegmentList, gPremises)); gPremises <math>\leftarrow nil; end;
```

1438. 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 1389\rangle += gThatPos: Position;
```

```
1439. \langle \text{Pop a proof step } 1434 \rangle + \equiv
```

```
itSupposeHead, itCaseHead, itAssumption: if gPremises \neq nil then begin gLastWSItem \uparrow .nContent \leftarrow new(CollectiveAssumptionPtr, Init(gThatPos, gPremises)); gPremises \leftarrow nil; end
```

```
else gLastWSItem \uparrow. nContent \leftarrow new(SingleAssumptionPtr, Init(nItemPos, new(PropositionPtr, Init(nLabel, gLastFormula, nPropPos))));
```

466 POPPING Mizar Parser $\S1440$

1440. Pop a conclusion or regular statement. We assign an appropriate WSM statement node to the previous item's contents.

```
\langle \text{Local variables for parser additions } 1389 \rangle + \equiv \\ gIterativeSteps: PList; \\ gIterativeLastFormula: FormulaPtr; \\ gInference: JustificationPtr; \\ \\ \textbf{1441.} \quad \langle \text{Pop a conclusion or regular statement } 1441 \rangle \equiv \\ \textbf{case } nRegularStatementKind } \textbf{of} \\ stDiffuseStatement: \\ nLastWSItem\uparrow.nContent \leftarrow new(DiffuseStatementPtr, Init(nLabel, stDiffuseStatement)); \\ stCompactStatement: nLastWSItem\uparrow.nContent \leftarrow new(CompactStatementPtr, Init(new(PropositionPtr, Init(nLabel, gLastFormula, nPropPos)), nInference)); \\ stIterativeEquality: nLastWSItem\uparrow.nContent \leftarrow new(IterativeEqualityPtr, Init(new(PropositionPtr, Init(nLabel, gIterativeLastFormula, nPropPos)), gInference, gIterativeSteps)); \\ \textbf{endcases}; \\ \text{This code is used in section 1430.} \\ \\
```

1442. Pop a 'let' statement. For generic let statements of the form

This code is used in section 1430.

```
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 1389\rangle + \equiv
gSuchPos: Position;
1443. \langle \text{Pop a "let" statement 1443} \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;
```

 $\S1444$ Mizar Parser POPPING 467

1444. 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 } 1444 \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 1445, 1446, 1447, and 1450.
This code is used in section 1430.
1445. 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 } 1444 \rangle + \equiv
itDefFunc: \mathbf{begin} \ nLastWSItem \uparrow .nContent \leftarrow new(FunctorDefinitionPtr, Init(qPatternPos,
       qRedefinitions, FunctorPatternPtr(qPattern), qSpecification, qDefiningWay, qDefiniens));
  end;
1446. 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 } 1444 \rangle + \equiv
itDefAttr: \mathbf{begin} \ nLastWSItem \uparrow .nContent \leftarrow new(AttributeDefinitionPtr, Init(gPatternPos,
       gRedefinitions, AttributePatternPtr(gPattern), gDefiniens));
  end;
1447. Pop a predicate definition. We just need to add a Predicate Definition object to the caller's
nLastWSItem's contents.
\langle \text{ Pop a definition item } 1444 \rangle + \equiv
itDefPred: \mathbf{begin} \ nLastWSItem \uparrow. nContent \leftarrow new(PredicateDefinitionPtr, Init(qPatternPos,
       gRedefinitions, PredicatePatternPtr(gPattern), gDefiniens));
  end;
1448. Popping a structure definition. We just need to add a StructureDefinition object to the caller's
nLastWSItem's contents.
\langle Global variables introduced in parseraddition.pas 1382\rangle + \equiv
qConstructorNr: integer;
1449. \langle Local variables for parser additions 1389\rangle + \equiv
qParams: PList;
gStructFields: PList;
1450. \langle \text{Pop a definition item } 1444 \rangle + \equiv
```

itDefStruct: **begin** $nLastWSItem \uparrow .nContent \leftarrow new(StructureDefinitionPtr, Init(qPatternPos,$

gStructPrefixes, gConstructorNr, gParams, gStructFields));

end;

468 POPPING Mizar Parser $\S1451$

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 $(\S1461)$ or similar methods. The gClusterTerm is populated in the extItemObj.FinishClusterTerm method (§1462). $\langle \text{Local variables for parser additions } 1389 \rangle + \equiv$ gAntecedent, gConsequent: PList; gClusterTerm: TermPtr;**1452.** $\langle \text{ Pop a registration item } 1452 \rangle \equiv$ 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 1455. This code is used in section 1430. 1453. 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 1382 $\rangle + \equiv$ gLeftTermInReduction: TermPtr;**1454.** $\langle \text{Local variables for parser additions 1389} \rangle + \equiv$ gIdentifyEqLociList: PList;**1455.** $\langle \text{Pop a registration item } 1452 \rangle + \equiv$ $itIdentify: \textbf{begin} \ nLastWSItem \uparrow. nContent \leftarrow new (IdentifyRegistrationPtr, Init(nItemPos, gNewPattern, and all the properties of the$ qPattern, qIdentifyEqLociList); include(qCorrectnessConditions, syCompatibility); end: itReduction: begin $nLastWSItem \uparrow .nContent \leftarrow new(ReduceRegistrationPtr, Init(nItemPos,$ qLeftTermInReduction, qLastTerm); include(qCorrectnessConditions, syReducibility);

 $itPropertyRegistration: SethoodRegistrationPtr(nLastWSItem \uparrow. nContent) \uparrow. nJustification \leftarrow nInference;$

end:

 $\{1456$ Mizar Parser POPPING 469

1456. 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 nLinkAllowed depending 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 1456 \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 1429.
```

Subsection 23.2.3. Registrations and notations

end:

1457. Processing synonyms. We need to update the *gNewPatternPos* and *gNewPattern* global variables when processing a synonym.

```
define process\_notation\_item \equiv qNewPatternPos \leftarrow qPatternPos; qNewPattern \leftarrow qPattern
\langle Extended item implementation 1406\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 (§1898). The
gAttrColl is populated in the extSubexpObj.CompleteAdjectiveCluster (§1578) method.
\langle \text{Local variables for parser additions } 1389 \rangle + \equiv
gAttrColl: PList;
1459. \langle Extended item implementation 1406 \rangle + \equiv
procedure extItemObj.StartAttributes;
  begin gAttrColl \leftarrow new(PList, Init(6));
```

1460. Starting a sentence. We just need to populate the caller's nPropPos, assigning to it the current position of the Parser.

```
⟨Extended item implementation 1406⟩ +≡ procedure extItemObj.StartSentence; begin nPropPos ← CurPos; end;
```

procedure *extItemObj* . *FinishClusterTerm*;

variable names for the subexpressions):

1461. 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 1406⟩ +≡
procedure extItemObj.FinishAntecedent;
begin gClusterSort ← ConditionalRegistration; gAntecedent ← gAttrColl;
end;
procedure extItemObj.FinishConsequent;
begin gConsequent ← gAttrColl;
end;
1462. Finishing a cluster. This populates the gClusterSort and the gClusterTerm.
⟨ Extended item implementation 1406⟩ +≡
```

begin $qClusterSort \leftarrow FunctorialRegistration; <math>qClusterTerm \leftarrow qLastTerm;$

end;1463. Identify registration. Schematically, we have the registration statement look like (using global)

```
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 \text{Extended item implementation } 1406 \rangle + \equiv \\ \textbf{procedure } extItemObj.StartFuncIdentify; \\ \textbf{begin end;} \\ \textbf{procedure } extItemObj.ProcessFuncIdentify; \\ \textbf{begin } gNewPatternPos \leftarrow gPatternPos; \ gNewPattern \leftarrow gPattern; \\ \textbf{end;} \\ \textbf{procedure } extItemObj.CompleteFuncIdentify; \\ \textbf{begin } gIdentifyEqLociList \leftarrow \textbf{nil}; \\ \textbf{if } CurWord.Kind = sy\_When \ \textbf{then } gIdentifyEqLociList \leftarrow new(PList,Init(0)); \\ \textbf{end:} \\ \end{aligned}
```

1464. "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 (§1503).

```
\langle Local variables for parser additions 1389\rangle +\equiv gLeftLocus: LocusPtr;
```

```
1465. ⟨Extended item implementation 1406⟩ +≡
procedure extItemObj.ProcessLeftLocus;
begin gLeftLocus ← 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 ← gPatternPos; gLeftTermInReduction ← gLastTerm;
end;
```

Subsection 23.2.4. Processing definitions

1466. The terminology used by the Parser appears to be ($\S\S1782 \ et \ seq.$):

```
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 1406\rangle +\equiv procedure extItemObj.StartFixedVariables; begin gQualifiedSegmentList \leftarrow new(PList, Init(0)); end;
```

```
1467. \langle Global variables introduced in parseraddition.pas 1382 \rangle +\equiv gQualifiedSegment: MList; gSegmentPos: Position;
```

1468. 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 1406\rangle += procedure extItemObj.StartFixedSegment; begin gQualifiedSegment.Init(0); gSegmentPos \leftarrow CurPos; end;
```

1469. When parsing fixed variables, and the Parser has just entered the loop to parse fixed variables, this function will be invoked.

```
⟨ Extended item implementation 1406⟩ +≡
procedure extItemObj.ProcessFixedVariable;
begin gQualifiedSegment.Insert(new(VariablePtr, Init(CurPos, GetIdentifier)));
end;
```

1470. This "clears the cache" for assigning the type in an explicitly qualified segment (appearing in a fixed variable segment).

```
⟨Extended item implementation 1406⟩ +≡ procedure extItemObj.ProcessBeing; begin gLastType ← nil; end;
```

1471. 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.

```
 \langle \text{Extended item implementation } 1406 \rangle + \equiv \\ \textbf{procedure } extItemObj.FinishFixedSegment; \\ \textbf{var } k: integer; \\ \textbf{begin if } gLastType \neq \textbf{nil then} \quad \{\text{explicitly qualified case}\} \\ \textbf{begin } gQualifiedSegmentList \uparrow.Insert(new(ExplicitlyQualifiedSegmentPtr, Init(gSegmentPos, new(PList, MoveList(gQualifiedSegment)), gLastType))); } gQualifiedSegment.DeleteAll; \\ \textbf{end} \\ \textbf{else begin for } k \leftarrow 0 \textbf{ to } gQualifiedSegment.Count - 1 \textbf{ do} \\ \textbf{begin } gQualifiedSegmentList \uparrow.Insert(new(ImplicitlyQualifiedSegmentPtr, Init(VariablePtr(gQualifiedSegment.Items \uparrow [k]) \uparrow.nVarPos, gQualifiedSegment.Items \uparrow [k]))); \\ gQualifiedSegment.Items \uparrow [k] \leftarrow \textbf{nil}; \\ \textbf{end}; \\ \textbf{end}; \\ gQualifiedSegment.Done; \\ \textbf{end}; \end{aligned}
```

1472. 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 (§1442) the gSuchPos is used when popping a let statement.

```
\langle Local \text{ variables for parser additions } 1389 \rangle + \equiv gSuchThatOcc: boolean; { not used }
```

```
1473. ⟨Extended item implementation 1406⟩ +≡ procedure extItemObj.FinishFixedVariables; begin gSuchThatOcc ← CurWord.Kind = sy_Such; gSuchPos ← CurPos; gPremises ← nil; end;
```

1474. 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 Extended item implementation 1406\rangle +\equiv procedure extItemObj.StartAssumption; begin gPremises \leftarrow new(PList,Init(0)); gThatPos \leftarrow CurPos; end:
```

1475. Finishing an assumption will update the global variable gBlockPtr's field reflecting it has assumptions.

```
⟨ Extended item implementation 1406⟩ +≡
procedure extItemObj.FinishAssumption;
begin ExtBlockPtr(gBlockPtr)↑.nHasAssumptions ← true;
end;
```

1476. 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 Extended item implementation 1406\rangle +\equiv procedure extItemObj.StartCollectiveAssumption; begin <math>gPremises \leftarrow new(PList, Init(0)); gThatPos \leftarrow CurPos; end;
```

Mizar Parser

1477. 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 } 1382\,\rangle\,+\!\equiv\,gDefLabId\colon\,integer;\,gDefLabPos\colon\,Position;
```

1478. $\langle \text{Local variables for parser additions 1389} \rangle + \equiv gOtherwise: PObject;$

```
1479. ⟨Extended item implementation 1406⟩ +≡
procedure extItemObj.ProcessMeans;
begin gDefLabId ← 0; gDefLabPos ← CurPos; gDefiningWay ← dfMeans; gOtherwise ← nil;
gMeansPos ← CurPos
```

procedure extItemObj.ProcessEquals;

end;

```
begin gDefLabId \leftarrow 0; gDefLabPos \leftarrow CurPos; gDefiningWay \leftarrow dfEquals; gOtherwise \leftarrow \mathbf{nil}; gMeansPos \leftarrow CurPos; end:
```

1480. 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 1406\rangle +\equiv procedure extItemObj.FinishOtherwise; begin if gDefiningWay = dfEquals then gOtherwise \leftarrow gLastTerm else gOtherwise \leftarrow gLastFormula; end;
```

§1481 Mizar Parser PROCESSING DEFINITIONS 475

1481. 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 1406⟩ +≡ procedure extItemObj.StartDefiniens; begin it_Allowed ← nItAllowed; end;
```

1482. "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 \, \text{Local variables for parser additions } \, 1389 \, \rangle \, + \equiv \, gPartDef \colon \, PObject;
```

```
1483. \langle \text{Extended item implementation } 1406 \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:
```

1484. After parsing a formula, then the Parser will invoke FinishGuard. This will append to gPartialDefs a new partial definiens.

```
\langle \text{ Extended item implementation } 1406 \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}
```

1485. 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 1406\rangle +\equiv procedure extItemObj.FinishSpecification; begin gSpecification \leftarrow gLastType; end;
```

1486. "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 1406\rangle +\equiv procedure extItemObj.FinishConstructionType;
```

```
begin gSpecification \leftarrow gLastType; end;
```

1487. 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 1406\rangle +\equiv procedure extItemObj.StartExpansion; begin if gRedefinitions then ErrImm(271); nDefiniensProhibited \leftarrow true; gExpandable \leftarrow true; end;
```

1488. The Parser, when determining the pattern for an attribute (§1845), 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 Global variables introduced in parseraddition.pas 1382 \rangle += gParamNbr:\ integer;
```

```
1489. \langle \text{Local variables for parser additions 1389} \rangle + \equiv gLocus: LocusPtr;
```

```
1490. \langle Extended item implementation 1406\rangle +\equiv procedure extItemObj.StartAttributePattern;
```

```
begin gParamNbr \leftarrow 0; gParams \leftarrow nil; gLocus \leftarrow new(LocusPtr, Init(CurPos, GetIdentifier)); end;
```

1491. 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 } 1406 \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}
```

1492. A mode definition may include a "sethood" property. This particular function is used when registering sethood in a registration block.

```
\langle Extended item implementation 1406\rangle +\equiv procedure extItemObj.FinishSethoodProperties; begin nLastWSItem\uparrow.nContent \leftarrow new(SethoodRegistrationPtr, Init(nItemPos, gPropertySort, gLastType)); end;
```

1493. 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 1406\rangle +\equiv procedure extItemObj.StartModePattern; begin gParamNbr \leftarrow 0; gParams \leftarrow \mathbf{nil}; gPatternPos \leftarrow CurPos; gConstructorNr \leftarrow CurWord.Nr; end;
```

1494. 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 } 1406 \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}
```

1495. When Parser starts parsing a new predicate pattern, we should reset the relevant global variables.

```
\langle Extended item implementation 1406\rangle +\equiv procedure extItemObj.StartPredicatePattern; begin gParamNbr \leftarrow 0; gParams \leftarrow nil; end;
```

1496. 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 1382\rangle +\equiv gLeftLociNbr: integer;

1497. \langle Local variables for parser additions 1389\rangle +\equiv gLeftLoci: PList;

1498. \langle Extended item implementation 1406\rangle +\equiv
```

procedure extItemObj.ProcessPredicateSymbol; **begin** $gPatternPos \leftarrow CurPos;$ $gLeftLociNbr \leftarrow gParamNbr;$ $gLeftLoci \leftarrow gParams;$ $gParamNbr \leftarrow 0;$ $gParams \leftarrow \mathbf{nil};$ $gConstructorNr \leftarrow CurWord.Nr;$ **end**;

1499. 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 } 1406 \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}
```

1500. 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.

```
1501. ⟨Extended item implementation 1406⟩ +≡

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;
```

§1502 Mizar Parser PROCESSING DEFINITIONS 479

1502. 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).

```
\label{eq:continuous} \begin{split} &\langle \operatorname{Extended item implementation } 1406 \rangle + \equiv \\ & \mathbf{procedure} \ extItemObj.ProcessFunctorSymbol; \\ & \mathbf{begin} \ gPatternPos \leftarrow CurPos; \\ & \mathbf{if} \ CurWord.Kind = InfixOperatorSymbol \ \mathbf{then} \\ & \mathbf{begin} \ gSubItemKind \leftarrow InfixOperatorSymbol; \ gConstructorNr \leftarrow CurWord.Nr; \\ & gLeftLociNbr \leftarrow gParamNbr; \ gLeftLoci \leftarrow gParams; \ gParamNbr \leftarrow 0; \ gParams \leftarrow \mathbf{nil}; \\ & \mathbf{end}; \\ & \mathbf{end}; \end{split}
```

1503. 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 Extended item implementation 1406\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 1406\rangle + \equiv
procedure extItemObj.StartVisible;
  begin gParams \leftarrow new(PList, Init(0));
  end:
```

 $gParams \leftarrow \mathbf{nil};$

end:

1505. The Parser iteratively calls its GetVisible (§1835) 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 1406⟩ +≡
procedure extItemObj.ProcessVisible;
begin inc(gParamNbr);
if gParams ≠ nil then gParams↑.Insert(new(LocusPtr, Init(CurPos, GetIdentifier)));
end;
```

1506. 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 1406⟩ +≡

procedure extItemObj.FinishPrefix;

begin gStructPrefixes.Insert(gLastType);
end;

1507. ⟨Extended item implementation 1406⟩ +≡

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;
```

1508. 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.

```
gFieldsNbr: integer;

1509. ⟨Extended item implementation 1406⟩ +≡

procedure extItemObj.StartFields;

var lFormatNr: integer;
```

 \langle Global variables introduced in parseraddition.pas 1382 $\rangle + \equiv$

begin $lFormatNr \leftarrow gFormatsColl.CollectPrefixForm(`L`, gConstructorNr, gParamNbr); in_AggrPattern <math>\leftarrow$ true; $gStructFields \leftarrow new(PList, Init(0)); gFieldsNbr \leftarrow 0;$ end;

1510. The Parser has just encountered the end structure bracket ("#)") token, so we want to add the format to the qFormatsColl dictionary.

```
⟨ Extended item implementation 1406⟩ +≡
procedure extItemObj.FinishFields;
var lFormatNr: integer;
begin lFormatNr ← gFormatsColl.CollectPrefixForm(´G´, gConstructorNr, gFieldsNbr);
end;
```

§1511 Mizar Parser PROCESSING DEFINITIONS 481

1511. 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 } 1389 \rangle + \equiv
gStructFieldsSegment: PList;
gSqmPos: Position;
1512. \langle Extended item implementation 1406 \rangle + \equiv
procedure extItemObj.StartAggrPattSegment;
  begin gStructFieldsSegment \leftarrow new(Plist, Init(0)); gSgmPos \leftarrow CurPos;
  end;
1513. For each selector-symbol the Parser encounters, it invokes the ProcessField.
\langle Extended item implementation 1406\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;
1514. 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 1406\rangle + \equiv
procedure extItemObj.FinishAggrPattSegment;
```

Subsection 23.2.5. Processing remaining statements

end:

1515. Processing schemes. Most of these methods are used in parsing a scheme block (§1915). It will be useful to examine that function to see where these methods are invoked.

 $\mathbf{begin}\ gStructFields.Insert(new(FieldSegmentPtr,Init(gSgmPos,gStructFieldsSegment,gLastType)));$

When the Parser starts a new scheme, several state variables need to be reset. The gSchemeIdNr is populated by the GetIdentifier (§1381) procedure, the gSchemeIdPos is assigned the current position, and the gSchemeParams should be allocated to an empty list.

```
\langle Extended item implementation 1406\rangle +\equiv procedure extItemObj.ProcessSchemeName; begin gSchemeIdNr \leftarrow GetIdentifier; <math>gSchemeIdPos \leftarrow CurPos; gSchemeParams \leftarrow new(PList,Init(0)); end;
```

1516. 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 1382 \rangle += gTypeList\colon MList;
```

```
1517. \langle \text{Extended item implementation } 1406 \rangle + \equiv procedure extItemObj.StartSchemeQualification; begin gSubItemKind \leftarrow CurWord.Kind; gTypeList.Init(4); end:
```

1518. 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 1382\rangle +\equiv gSubItemPos: Position;
```

```
1519. \langle Extended item implementation 1406\rangle +\equiv procedure extItemObj.FinishSchemeQualification; begin gSubItemPos \leftarrow CurPos end;
```

1520. 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 1382 \rangle += gSchVarIds\colon MList;
```

```
1521. \langle Extended item implementation 1406 \rangle + \equiv procedure extItemObj.StartSchemeSegment; begin gSubItemPos \leftarrow CurPos; <math>gSchVarIds.Init(2); end;
```

1522. 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 1406⟩ +≡
procedure extItemObj.ProcessSchemeVariable;
begin gSchVarIds.Insert(new(VariablePtr, Init(CurPos, GetIdentifier)));
end;
```

1523. 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
```

1524. 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 1406\rangle +\equiv procedure extItemObj.FinishSchemeThesis; begin <math>gSchemeConclusion \leftarrow gLastFormula; <math>gSchemePremises \leftarrow new(Plist,Init(0)); end;
```

1525. 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 Extended item implementation 1406\rangle +\equiv procedure extItemObj.FinishSchemePremise; begin gSchemePremises \uparrow.Insert(new(PropositionPtr, Init(nLabel, gLastFormula, nPropPos))); end;
```

1526. Reserved variables. These methods are invoked only when the Parser parses a reservation (§1911). 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 1382 \rangle + \equiv
qResIdents: PList;
qResPos: Position;
1527. \langle Extended item implementation 1406 \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;
```

1528. 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]").
⟨ Global variables introduced in parseraddition.pas 1382⟩ +≡
gPrivateId: Integer;
gPrivateIdPos: Position;

1529. ⟨ Extended item implementation 1406⟩ +≡
procedure extItemObj.StartPrivateDefiniendum;
begin gPrivateId ← GetIdentifier; gPrivateIdPos ← CurPos; dol_Allowed ← true; gTypeList.Init(4);
end;
```

1530. 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 1406⟩ +≡
procedure extItemObj.FinishLocusType;
begin gTypeList.Insert(gLastType);
end;
```

1531. 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 1406\rangle +\equiv procedure extItemObj.CreateExpression(fExpKind: ExpKind); begin <math>gExpPtr \leftarrow new(extExpressionPtr, Init(fExpKind)); end:
```

1532. 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 } 1406 \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}
```

1533. 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 *qPrivateIdPos* state variable to be the current position.

```
⟨ Extended item implementation 1406⟩ +≡
procedure extItemObj.StartPrivateConstant;
begin gPrivateId ← GetIdentifier; gPrivateIdPos ← CurPos;
end;
```

1534. 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).

```
⟨ Extended item implementation 1406⟩ +≡ procedure extItemObj.StartPrivateDefiniens; begin dol_Allowed ← 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 1406\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;
1536. When finishing the definiendum formula for a "defpred", the Parser invokes this FinishPrivatePredDefinienition
method.
\langle Extended item implementation 1406\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;
1537.
                                   Reconsider statements.
\langle Extended item implementation 1406\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:
1538. This is invoked when parsing a private item which is a "reconsider" statement.
\langle Extended item implementation 1406\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:
1539. 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 1406\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:
```

1540. In statements of the form

```
\mathtt{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 1406\rangle +\equiv procedure extItemObj.FinishHypothesis; begin if <math>gPremises \neq nil then gPremises \uparrow.Insert(new(PropositionPtr,Init(nLabel,gLastFormula,nPropPos))); end;
```

1541. "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 1406\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}
```

end;

```
1542. In statements of the form
```

```
take \langle Term \rangle;
the Parser begins by invoking StartExemplifyingTerm, parses the term, then FinishExemplifyingTerm.
\langle Extended item implementation 1406\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;
```

1543. When the Parser examines the correctness conditions (§1869), 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 1406\rangle +\equiv procedure extItemObj.ProcessCorrectness; begin if CurWord.Kind \neq sy\_Correctness then if (gCorrectnessConditions \neq []) \land \neg AxiomsAllowed then Error(gDefPos, 73); end:
```

1544. 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 1406\rangle +\equiv procedure extItemObj.StartConstructionType; begin if gRedefinitions \land (CurWord.Kind = sy\_Arrow) then include(gCorrectnessConditions, syCoherence); end;
```

1545. 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.

```
\langle Extended item implementation 1406\rangle +\equiv procedure extItemObj.ProcessLabel; begin nLabelIdNr \leftarrow 0; nLabelIdPos \leftarrow CurPos; if (CurWord.Kind = Identifier) \wedge (AheadWord.Kind = sy\_Colon) then nLabelIdNr \leftarrow CurWord.Nr; nLabel \leftarrow new(LabelPtr, Init(nLabelIdNr, nLabelIdPos)); end;
```

end;

end:

1546. A regular statement is either a "diffuse" statement (which occurs with the "now" keyword) or else it's a "compact" statement.

```
\langle Extended item implementation 1406\rangle +\equiv procedure extItemObj.StartRegularStatement; begin if CurWord.Kind = sy\_Now then nRegularStatementKind \leftarrow stDiffuseStatement else nRegularStatementKind \leftarrow stCompactStatement; end:
```

1547. 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 1382\rangle +\equiv gDefLabel: LabelPtr;

1548. \langle Extended item implementation 1406\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));
```

1549. 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 1406⟩ +≡
procedure extItemObj.ProcessSchemeReference;
begin if CurWord.Kind = Identifier then
   begin SchemeJustificationPtr(nInference)↑.nSchemeIdNr ← CurWord.Nr;
   SchemeJustificationPtr(nInference)↑.nSchemeInfPos ← CurPos;
   end;
end;
```

1550. 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 1382⟩ +≡
gTHEFileNr: integer;

1551. ⟨Extended item implementation 1406⟩ +≡
procedure extItemObj.StartLibraryReferences;
begin gTHEFileNr ← CurWord.Nr;
```

1552. 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 1406\rangle +\equiv procedure extItemObj.StartSchemeLibraryReference; begin gTHEFileNr \leftarrow CurWord.Nr; end;
```

with $SchemeJustificationPtr(nInference) \uparrow do$

end; end;

1553. For references to labels found in the article being processed ("private references"), this method is invoked. \langle Extended item implementation 1406 $\rangle + \equiv$ **procedure** extItemObj.ProcessPrivateReference; **begin** $SimpleJustificationPtr(nInference)\uparrow.nReferences\uparrow.Insert(new(LocalReferencePtr,$ Init(GetIdentifier, CurPos))); end; 1554. 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 1382 $\rangle + \equiv$ *qDefinitional*: boolean; **1555.** \langle Extended item implementation $1406\rangle + \equiv$ **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, **1556.** 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 1406 $\rangle + \equiv$ **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 1406 $\rangle + \equiv$ **procedure** extItemObj.ProcessSchemeNumber; **begin if** $CurWord.Kind \neq Numeral$ **then** exit; if CurWord.Nr = 0 then **begin** ErrImm(146); exitend:

begin $nSchFileNr \leftarrow gTHEFileNr$; $nSchemeIdNr \leftarrow CurWord.Nr$; $nSchemeInfPos \leftarrow PrevPos$;

end:

end;

1558. This appears when the Parser starts its *Justification* ($\S1803$) procedure, or in the *Regular Statement* ($\S1832$) procedure.

This clears the *nInference*, reassigning it to the **nil** pointer.

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 1406\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;
1559. A simple justification is either a Scheme-Justification ("from..."), a Straightforward-Justification
("by..."), or... somethign else?
\langle Extended item implementation 1406\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;
1560. 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))
\langle Extended item implementation 1406\rangle + \equiv
procedure extItemObj.FinishSimpleJustification;
  begin with extBlockPtr(gBlockPtr)\uparrow do
```

if $(nInference \uparrow .nInfSort = infStraightforwardJustification) \lor (nInference \uparrow .nInfSort = infError)$ then

begin if $is_inference_error$ **then** $nInference \uparrow .nInfSort \leftarrow infError$;

 $extBlockPtr(gBlockPtr)\uparrow.nLinked \leftarrow false;$

1561. 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 1406\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;
```

1562. 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 1389\rangle += gIterPos: Position;
```

```
1563. ⟨Extended item implementation 1406⟩ +≡ procedure extItemObj.StartIterativeStep; begin gIterPos ← CurPos; end;
```

1564. 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 1406⟩ +≡
procedure extItemObj.FinishIterativeStep;
begin gIterativeSteps↑.Insert(new(IterativeStepPtr, Init(gIterPos, gLastTerm, nInference)));
end:
```

1565. 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 1406\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

1566. 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.

1567. 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

```
\label{eq:condition} \begin{tabular}{ll} for $\langle Variables \rangle$ st $\langle Restriction \rangle$ holds $\dots$ \\ \\ define $arg\_type \equiv {\bf record} \ Start, Length: integer; \\ & {\bf end} \\ \\ define $func\_type \equiv {\bf record} \ Instance, SymPri: integer; \\ & FuncPos: \ Position; \\ & {\bf end} \\ \\ \langle {\bf Methods implemented by subclasses of} \ SubexpObj \ 876 \rangle + \equiv \\ \end{tabular}
```

TermNbr: integer;

```
1568.
        \langle Extended subexpression class declaration 1568\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 (§1714) }
    nArgList: array of arg_type;
    nFunc: array of func_type;
    constructor Init;
    \langle Methods implemented by subclasses of SubexpObj 876\rangle
  end;
This code is used in section 1379.
1569. 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 } 1389 \rangle + \equiv
```

Mizar Parser

end:

```
1570. \langle \text{Extended subexpression implementation } 1570 \rangle \equiv \{ \text{Subexpressions handling} \}
constructor extSubexpObj.Init;
const MaxArgListNbr = 20;
begin inherited\,Init;\,\,nRestriction \leftarrow \mathbf{nil};\,\,nTermBase \leftarrow TermNbr;\,\,nArgListNbr \leftarrow 0;
setlength\,(nArgList,\,MaxArgListNbr + 1);\,\,setlength\,(nFunc,\,MaxArgListNbr + 1);
nArgList\,[0].Start \leftarrow TermNbr + 1;
```

```
See also sections 1571, 1572, 1573, 1575, 1576, 1577, 1578, 1579, 1580, 1581, 1582, 1583, 1584, 1585, 1586, 1587, 1588, 1589, 1590, 1591, 1593, 1597, 1609, 1610, 1611, 1612, 1613, 1614, 1615, 1616, 1617, 1618, 1619, 1620, 1621, 1622, 1623, 1624, 1625, 1626, 1627, 1628, 1629, 1630, 1631, 1632, 1633, 1634, 1635, 1636, 1637, 1638, 1639, 1640, 1641, 1642, 1643, 1644, 1645, 1646, 1647, 1648, 1649, 1650, 1651, 1652, 1653, 1654, 1655, 1656, 1657, 1658, 1659, 1660, 1661, 1662, 1663, 1664, 1665, 1666, 1667, and 1668.
```

This code is used in section 1380.

1571. 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 1570\rangle +\equiv procedure extSubexpObj.StartAttributes; begin nAttrCollection.Init(0); gLastType \leftarrow nil; end:
```

1572. 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 1570\rangle +\equiv procedure extSubexpObj.ProcessNon; begin nNoneOcc \leftarrow CurWord.Kind = sy\_Non; nNonPos \leftarrow CurPos; end;
```

1573. 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.

```
⟨ Extended subexpression implementation 1570⟩ +≡ 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;
```

1574. The "process (singular) attribute" method is invoked in the "process (plural) attributes" procedure ($\S1725$), and in the ATTSubexpression procedure ($\S1895$). 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 1382 \rangle +\equiv gLastAdjective: AdjectiveExpressionPtr;
```

 \langle Extended subexpression implementation 1570 \rangle + \equiv **procedure** extSubexpObj.CompleteAdjectiveCluster;

end:

begin $gAttrColl \leftarrow new(PList, MoveList(nAttrCollection));$

```
1575.
        \langle Extended subexpression implementation 1570 \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:
1576.
        These next next method is invoked before the Parser parses arguments for an attribute.
\langle Extended subexpression implementation 1570 \rangle + \equiv
procedure extSubexpObj.StartAttributeArguments;
  begin nTermBase \leftarrow TermNbr;
  end;
1577.
        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 1570 \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.
```

498

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 Register Cluster (§1898) procedure.

```
\langle Extended subexpression implementation 1570 \rangle + \equiv
procedure extSubexpObj.CompleteClusterTerm;
  begin if TermNbr - nTermBase > 1 then
    begin ErrImm(379); gLastTerm \leftarrow new(IncorrectTermPtr, Init(CurPos));
    end;
  end;
```

1580. A "simple term" appears to be a variable. This is used when the Parser parses an identifier as a closed term ($\S1696$). The state variable gLastTerm is updated to point to a newly allocated SimpleTerm AST node ($\S916$).

This method should probably be moved closer to the other methods used when parsing terms.

```
\langle Extended subexpression implementation 1570 \rangle + \equiv
procedure extSubexpObj.ProcessSimpleTerm;
  begin gLastTerm \leftarrow new(SimpleTermPtr, Init(CurPos, GetIdentifier));
  end;
```

Qualified terms. The Parser invokes *ProcessQua* when it is looking directly at a "qua" token, 1581. specifically in the AppendQua (§1689) 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.

```
\langle Extended subexpression implementation 1570 \rangle + \equiv
{\bf procedure}\ {\it extSubexpObj.ProcessQua};
  begin nQuaPos \leftarrow CurPos
  end;
```

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 qLastType, 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 (§1689) procedure.

```
\langle Extended subexpression implementation 1570 \rangle + \equiv
procedure extSubexpObj.FinishQualifiedTerm;
  begin Term[TermNbr] \leftarrow new(QualifiedTermPtr, Init(nQuaPos, Term[TermNbr], qLastType));
  end;
```

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 Exactly Term object.

```
\langle Extended subexpression implementation 1570 \rangle + \equiv
procedure extSubexpObj.ProcessExactly;
  begin nQuaPos \leftarrow CurPos; Term[TermNbr] \leftarrow new(ExactlyTermPtr, Init(nQuaPos, Term[TermNbr]));
  end:
```

1584. Arguments to a term. The *CheckTermLimit* procedure is a "private helper function" for the *FinishArgument* method.

```
⟨ Extended subexpression implementation 1570⟩ +≡
procedure CheckTermLimit;
var l: integer;
begin if TermNbr ≥ length(Term) then
begin l ← 2 * length(Term); setlength(Term,l);
end;
end;
```

1585. 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 1570\rangle +\equiv procedure extSubexpObj.FinishArgument; begin CheckTermLimit; inc(TermNbr); Term[TermNbr] \leftarrow gLastTerm; end:
```

1586. 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 (§1714).
```

This should probably check that the Term stack is not empty before being invoked.

```
\langle Extended subexpression implementation 1570 \rangle + \equiv procedure extSubexpObj.FinishTerm; begin gLastTerm \leftarrow Term[TermNbr]; dec(TermNbr); end;
```

Subsection 23.3.1. Parsing Types

1587. 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 1570⟩ +≡
procedure extSubexpObj.StartType;
begin gLastType ← nil;
end;
```

1588. This is invoked only by the Parser's RadixTypeSubexpression (§1727) 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 } 1570 \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}
```

500 Parsing types Mizer Parser $\S1589$

1589. 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 (§1727) procedure.

```
\langle Extended subexpression implementation 1570 \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;
```

1590. 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 1570\rangle +\equiv procedure extSubexpObj.InsertIncorrType; begin gLastType \leftarrow new(IncorrectTypePtr, Init(CurPos)); end;
```

 $\S1591$ Mizar Parser PARSING TYPES 501

1591. 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 } 1570 \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

1592. 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 ($\S808$) global variable, comes into play.

1593. 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 1570\rangle +\equiv procedure extSubexpObj.StartLongTerm; begin nArgListNbr \leftarrow 0; nArgList[0].Length \leftarrow TermNbr - nTermBase; end;
```

1594. 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;
```

1595. 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.

1596. 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.

1597. The only two place where *FinishLongTerm* is invoked is in the *AppendFunc* procedure (§1714) in parser.pas.

This relies on MFormatsList.LookUpFuncFormat (§827), which attempts to look up an MInfixFormatObj (§815) 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 ($\S1567$) that nArgList is an array of "**record** Instance, SymPri: integer; FuncPos: Position; end".

```
\langle Extended subexpression implementation 1570 \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 1606)
  label Corrected, AfterBalance;
  begin (Rebalance the long term tree 1598)
AfterBalance: (Construct the term's syntax tree after balancing arguments among subterms 1607)
  end:
```

Mizar Parser

See also section 1602.

This code is used in section 1598.

PARSING OPERATOR PRECEDENCE

1598. Rebalancing the term tree.

```
Note that nArgListNbr is mutated only in extSubexpObj.ProcessFunctorSymbol (§1609), and in ProcessAtomicFormula
(\S 1639) it is reset to zero.
  define missing\_functor\_format \equiv qFormatsColl.LookUp\_FuncFormat(Instance, l, r) = 0
\langle Rebalance the long term tree 1598\rangle \equiv
  (Initialize To_Right and ArgsLength arrays 1601)
  (Initialize Bl, goto AfterBalance if term has at most one argument 1603)
    \{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 1604 \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 1599)
  (Check for 174/175 error, goto AfterBalance if erred 1600)
This code is used in section 1597.
1599. (Check for 172/173 error, goto AfterBalance if erred 1599) \equiv
  if missing_functor_format then
    begin Error(FuncPos, 172); Error(nFunc[nArgListNbr].FuncPos, 173); goto AfterBalance; end;
This code is used in section 1598.
1600. (Check for 174/175 error, goto AfterBalance if erred 1600) \equiv
  with nFunc[Bl], ArgsLength[Bl] do
    if missing_functor_format then
       begin Error (FuncPos, 174); Error (nFunc[nArqListNbr].FuncPos, 175); goto AfterBalance; end;
This code is used in section 1598.
1601. We first allocate the arrays, the we initialize the values.
\langle \text{Initialize } To\_Right \text{ and } ArgsLength \text{ arrays } 1601 \rangle \equiv
  setlength(ArgsLength, nArgListNbr + 1); setlength(To\_Right, nArgListNbr + 1);
  setlength(Depo, nArqListNbr + 1);
```

1602. 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 } 1601 \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;
```

1603. 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 1603⟩ ≡ 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 1598.

```
\langle \text{Guess the } k^{th} \text{ functor format } 1604 \rangle \equiv
1604.
  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 1605\rangle
    end:
  Bl \leftarrow new\_Bl;
  end:
This code is used in section 1598.
1605. Check term Bl has valid functor format, goto AfterBalance if not 1605 \ge 100
  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 1604.
1606. 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 } 1606 \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 1597.
```

```
1607.
         \langle Construct the term's syntax tree after balancing arguments among subterms 1607 \rangle \equiv
  \langle Initialize symbol priorities, determine last ll, pl values 1608\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); \{ (\S1573) \}
          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 1597.
1608. (Initialize symbol priorities, determine last ll, pl values 1608) \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 1607.
```

Subsection 23.3.3. Processing subexpressions

1609. Note that ProcessFunctorSymbol is the only place where nArgListNbr is incremented. Processing functor symbols occurs in the Parser's AppendFunc (§1714) in a loop.

```
\langle \text{ Extended subexpression implementation } 1570 \rangle + \equiv \\ \textbf{procedure } extSubexpObj.ProcessFunctorSymbol; \\ \textbf{var } l: integer; \\ \textbf{begin } inc(nArgListNbr); \\ \textbf{if } nArgListNbr \geq length(nFunc) \textbf{ then} \\ \textbf{begin } l \leftarrow 2*length(nFunc) + 1; setlength(nArgList,l); setlength(nFunc,l); \\ \textbf{end}; \\ nArgList[nArgListNbr].Start \leftarrow TermNbr + 1; nFunc[nArgListNbr].FuncPos \leftarrow CurPos; \\ nFunc[nArgListNbr].Instance \leftarrow CurWord.Nr; \\ \textbf{end}; \\ \end{aligned}
```

1610. 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 1570\rangle +\equiv procedure extSubexpObj.FinishArgList; begin nArgList[nArgListNbr].Length \leftarrow TermNbr - nArgList[nArgListNbr].Start + 1; end;
```

1611. 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 } 1570 \right\rangle + \equiv   \mathbf{procedure } \ extSubexpObj.StartFraenkelTerm;   \mathbf{begin } \ nSample \leftarrow gLastTerm;   \mathbf{end};
```

1612. This is only invoked in the Parser's *ProcessPostqualification* (§1691) procedure, which is only invoked after the Parser calls the *extSubexp* object's *StartFraenkelTerm* method.

```
⟨ Extended subexpression implementation 1570⟩ +≡
procedure extSubexpObj.StartPostqualification;
begin nPostQualList.Init(0);
end;
```

1613. 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 1570⟩ +≡
procedure extSubexpObj.StartPostQualifyingSegment;
begin nSegmentIdentColl.Init(2);
end;
```

1614. 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 1570⟩ +≡
procedure extSubexpObj.ProcessPostqualifiedVariable;
begin nSegmentIdentColl.Insert(new(VariablePtr, Init(CurPos, GetIdentifier)));
end;
```

1615. 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 1570⟩ +≡
procedure extSubexpObj.StartPostqualificationSpecyfication;
begin nSegmentPos ← CurPos; gLastType ← nil;
end;
```

1616. 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 1570 \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;
```

1617. 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 1570\rangle +\equiv procedure extSubexpObj.FinishFraenkelTerm; begin gLastTerm \leftarrow new(FraenkelTermPtr, Init(CurPos, new(PList, MoveList(nPostQualList)), nSample, gLastFormula)); end;
```

1618. 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 1570\rangle +\equiv procedure extSubexpObj.StartSimpleFraenkelTerm; begin nAllPos \leftarrow CurPos; end;
```

1619. 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 } 1570 \rangle + \equiv \\ \textbf{procedure } extSubexpObj.FinishSimpleFraenkelTerm; \\ \textbf{begin } gLastTerm \leftarrow new(SimpleFraenkelTermPtr, Init(nAllPos, new(PList, MoveList(nPostQualList)), nSample)); \\ \textbf{end;}
```

1620. 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).

```
\langle Extended subexpression implementation 1570\rangle +\equiv procedure extSubexpObj.StartPrivateTerm; begin nSubexpPos \leftarrow CurPos; nSpelling \leftarrow CurWord.Nr; end;
```

1621. 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 1570\rangle +\equiv procedure extSubexpObj.FinishPrivateTerm; 

begin <math>gLastTerm \leftarrow new(PrivateFunctorTermPtr, Init(nSubexpPos, nSpelling, CreateArgs(nTermBase + 1))); 

end;
```

1622. 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 1570\rangle += procedure extSubexpObj.StartBracketedTerm; begin nSymbolNr \leftarrow CurWord.Nr; end;
```

1623. 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 } 1570 \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;}
```

1624. 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 (§1698), it first invokes this method. This stores the ID number for the structure in the caller's nSymbolNr.

```
\langle Extended subexpression implementation 1570\rangle +\equiv procedure extSubexpObj.StartAggregateTerm; begin nSymbolNr \leftarrow CurWord.Nr; end:
```

1625. 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 1570\rangle += procedure extSubexpObj.FinishAggregateTerm; var lFormatNr: integer; begin lFormatNr \leftarrow gFormatsColl.LookUp\_PrefixFormat(`G`, nSymbolNr, TermNbr - nTermBase); if lFormatNr = 0 then Error(CurPos, 176); { missing format error } gLastTerm \leftarrow new(AggregateTermPtr, Init(CurPos, nSymbolNr, CreateArgs(nTermBase + 1))); end;
```

1626. The Parser is parsing for a closed subterm, and has stumbled across "the" and is looking at a selector token ($\S1706$). 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 1570\rangle +\equiv procedure extSubexpObj.StartSelectorTerm; begin nSymbolNr \leftarrow CurWord.Nr; nSubexpPos \leftarrow CurPos; nNextWord \leftarrow AheadWord.Kind; end;
```

1627. 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 } 1570 \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}
```

1628. The Parser is about to start parsing a forgetful functor ($\S1709$) — 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 1570\rangle +\equiv procedure extSubexpObj.StartForgetfulTerm; begin nSymbolNr \leftarrow CurWord.Nr; nSubexpPos \leftarrow CurPos; nNextWord \leftarrow AheadWord.Kind; end:
```

1629. 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 1570\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;
```

1630. 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 (§1706).
- (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 (§1709).
- (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" (§1711).

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 1570⟩ +≡
procedure extSubexpObj.StartChoiceTerm;
begin nSubexpPos ← CurPos;
end;
```

1631. 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 1570\rangle +\equiv procedure extSubexpObj.FinishChoiceTerm; begin gLastTerm \leftarrow new(ChoiceTermPtr,Init(nSubexpPos,gLastType)); end;
```

1632. When the Parser encounters a numeral while seeking a closed subterm ($\S1695$), 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 1570\rangle +\equiv procedure extSubexpObj.ProcessNumeralTerm; 
begin <math>gLastTerm \leftarrow new(NumeralTermPtr, Init(CurPos, CurWord.Nr)); end;
```

514

1633. The Parser tries to parse a closed subterm ($\S1695$) and encounters the "it" token. Well, if the $it_Allowed$ state variable is true, then we should allocate a new ItTerm and update the gLastTerm 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 1570\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;
```

1634. 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.

```
⟨ Extended subexpression implementation 1570⟩ +≡
procedure extSubexpObj.ProcessLocusTerm;
begin if dol_Allowed then gLastTerm ← new(PlaceholderTermPtr, Init(CurPos, CurWord.Nr))
else begin gLastTerm ← new(IncorrectTermPtr, Init(CurPos)); ErrImm(181)
end;
end;
```

1635. Calamity! An incorrect expression has crossed the Parser's path. Allocate an IncorrectTerm object located at the Parser's current position, then update the gLastTerm state variable to point to it.

```
\langle Extended subexpression implementation 1570\rangle +\equiv procedure extSubexpObj.InsertIncorrTerm; begin gLastTerm \leftarrow new(IncorrectTermPtr, Init(CurPos)); end;
```

Subsection 23.3.4. Parsing formulas

1636. The Parser is trying to parse an atomic formula ($\S1752$), 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 1570\rangle +\equiv procedure extSubexpObj.InsertIncorrBasic; begin gLastFormula \leftarrow new(IncorrectFormulaPtr, Init(CurPos)); TermNbr <math>\leftarrow nTermBase; end;
```

1637. 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 1570\rangle +\equiv procedure extSubexpObj.InsertIncorrFormula; 
begin <math>gLastFormula \leftarrow new(IncorrectFormulaPtr, Init(CurPos)); end:
```

 $\S1638$ Mizar Parser PARSING FORMULAS 515

1638. If we are in a proof, allocate a new *ThesisFormula* object (recall the WEB macro for this §1399). Otherwise, raise a 65 error.

```
\langle Extended subexpression implementation 1570\rangle +\equiv procedure extSubexpObj.ProcessThesis; begin if gProofCnt > 0 then gLastFormula \leftarrow thesis\_formula else begin ErrImm(65); gLastFormula \leftarrow new(IncorrectFormulaPtr, Init(CurPos)); end; end;
```

1639. The Parser has encountered " $\langle Term \rangle$ is", or some other generic atomic formula (§1752), 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 } 1570 \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; \\ \end{cases}
```

1640. The Parser is either finishing a "predicative formula" (§1751) or it's parsing a predicate pattern (§1853), 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 1570\rangle +\equiv procedure extSubexpObj.ProcessPredicateSymbol; begin <math>nSubexpPos \leftarrow CurPos; case CurWord.Kind of sy\_Equal, PredicateSymbol: <math>nSymbolNr \leftarrow CurWord.Nr; othercases nSymbolNr \leftarrow 0; endcases; nRightArgBase \leftarrow TermNbr; end:
```

end;

516 Parsing formulas Mizar Parser $\S1641$

The Parser is parsing a "predicate formula" which has arguments on the righthand side of the predicate symbol (§1746). \langle Extended subexpression implementation 1570 $\rangle + \equiv$ **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" (§1751), then this method is invoked to construct **1642.** 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 $1570 \rangle + \equiv$ **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; 1643. The Parser tries to construct an AST when finishing up the right-hand side of a predicative formula (§1746), it invokes this method after the extSubexpObj.FinishPredicativeFormula has been invoked. \langle Extended subexpression implementation 1570 $\rangle + \equiv$ **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" (§1747), 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 $1570 \rangle + \equiv$ **procedure** extSubexpObj.StartMultiPredicativeFormula; **begin** nMultiPredicateList.Init(4); nMultiPredicateList.Insert(qLastFormula); end;

§1645 Mizar Parser PARSING FORMULAS 517

1645. 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 } 1570 \rangle + \equiv \\ \textbf{procedure } extSubexpObj.FinishMultiPredicativeFormula;} \\ \textbf{begin } gLastFormula \leftarrow new(MultiPredicativeFormulaPtr, Init(nSubexpPos, new(PList, MoveList(nMultiPredicateList))));} \\ \textbf{end;} \\ \end{aligned}
```

1646. 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 gLastFormula 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 } 1570 \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{aligned}
```

1647. 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 Extended subexpression implementation 1570\rangle +\equiv procedure extSubexpObj.FinishAttributiveFormula; begin gLastFormula \leftarrow new(AttributiveFormulaPtr,Init(nSubExpPos,Term[TermNbr],new(PList,MoveList(nAttrCollection)))); if nPostNegated then gLastFormula \leftarrow new(NegativeFormulaPtr,Init(nNotPos,gLastFormula)); dec(TermNbr); end;
```

518 Parsing formulas Mizar Parser $\S1648$

1648. 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 1570\rangle +\equiv procedure extSubexpObj.StartPrivateFormula; 
begin <math>nTermBase \leftarrow TermNbr; nSubexpPos \leftarrow CurPos; nSpelling \leftarrow CurWord.Nr; end;
```

1649. 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 (§1573) owned by the formula object. The *gLastFormula* is updated to point to the newly allocated formula object.

```
\langle Extended subexpression implementation 1570\rangle +\equiv procedure extSubexpObj.FinishPrivateFormula; 
begin <math>gLastFormula \leftarrow new(PrivatePredicativeFormulaPtr, Init(nSubexpPos, nSpelling, CreateArgs(nTermBase + 1))); 
end;
```

1650. 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 1570\rangle +\equiv procedure extSubexpObj.ProcessContradiction; 
begin <math>gLastFormula \leftarrow new(ContradictionFormulaPtr, Init(CurPos)); end;
```

1651. 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 1570\rangle +\equiv procedure extSubexpObj.ProcessNegative; begin gLastFormula \leftarrow new(NegativeFormulaPtr, Init(CurPos, gLastFormula)); end;
```

1652. 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 1570 \rangle +\equiv procedure extSubexpObj.ProcessNegation; begin nPostNegated \leftarrow \neg nPostNegated; nNotPos \leftarrow CurPos; end;
```

1653. 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 1570\rangle +\equiv procedure extSubexpObj.ProcessBinaryConnective; 
begin <math>nConnective \leftarrow CurWord.Kind; nFirstSententialOperand \leftarrow gLastFormula; nSubexpPos \leftarrow CurPos; end;
```

 $\S1654$ Mizar Parser PARSING FORMULAS 519

```
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 1570\rangle + \equiv
procedure extSubexpObj.ProcessFlexDisjunction;
  begin nFirstSententialOperand \leftarrow gLastFormula;
  end;
1655.
        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 1570 \rangle + \equiv
procedure extSubexpObj.ProcessFlexConjunction;
  begin nFirstSententialOperand \leftarrow qLastFormula;
  end:
        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 1570 \rangle + \equiv
procedure extSubexpObj.StartRestriction;
  begin nRestrPos \leftarrow CurPos;
  end;
        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 1570 \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 1570 \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:
```

520 Parsing formulas Mizar Parser $\S1659$

1659. 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.

```
There is a comment in Polish, "polaczyc z flexConj", which Google translates to "connect to flexConj".

(Extended subexpression implementation 1570) +=

procedure extSubexpObj.FinishFlexDisjunction; { polaczyc z flexConj }

begin gLastFormula ← new(FlexaryDisjunctiveFormulaPtr, Init(CurPos, nFirstSententialOperand,
 gLastFormula));

end;
```

1660. 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.

```
 \langle \text{ Extended subexpression implementation } 1570 \rangle + \equiv \\ \textbf{procedure } extSubexpObj.FinishFlexConjunction; \\ \textbf{begin } gLastFormula \leftarrow new(FlexaryConjunctiveFormulaPtr,Init(CurPos,nFirstSententialOperand, \\ gLastFormula)); \\ \textbf{end;}
```

1661. 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 1570\rangle +\equiv procedure extSubexpObj.StartExistential; begin nQualifiedSegments.Init(0); <math>nSubexpPos \leftarrow CurPos; end;
```

1662. 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 1570\rangle +\equiv procedure extSubexpObj.StartUniversal; begin nQualifiedSegments.Init(0); <math>nSubexpPos \leftarrow CurPos; end;
```

1663. After the Parser has invoked *StartUniversal* or *StartExistential*, it parses the quantified variables (which begins by invoking this method).

```
\langle Extended subexpression implementation 1570\rangle +\equiv procedure extSubexpObj.StartQualifiedSegment; begin nSegmentIdentColl.Init(2); nSegmentPos \leftarrow CurPos; end;
```

1664. 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 1570⟩ +≡ procedure extSubexpObj.StartQualifyingType; begin gLastType ← nil; end;
```

 $\{1665$ Mizar Parser PARSING FORMULAS 521

1665. 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 ExplicitlyQualifiedSegment object.

In both cases, the new allocated *QuantifiedSegment* object is appended to the caller's *nQualifiedSegments* field

```
\langle \text{ Extended subexpression implementation } 1570 \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
```

1666. 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 1570⟩ +≡
procedure extSubexpObj.ProcessVariable;
begin nSegmentIdentColl.Insert(new(VariablePtr, Init(CurPos, GetIdentifier)));
end;
```

522 Parsing formulas Mizar Parser $\S 1667$

```
1667.
          The Parser has just finished something like
                       ex \langle Qualified\text{-}Variables \rangle, ..., \langle Qualified\text{-}Variables \rangle st \langle Formula \rangle
Now we assemble it as
               ex \langle Qualified\text{-}Variables \rangle \text{ st } (ex \ldots \text{ st } (ex \langle Qualified\text{-}Variables \rangle \text{ st } \langle Formula \rangle))
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-Qualified-Variables \rangle
                     \langle Explicitly-Qualified-Variables \rangle
                    | \( Explicitly-Qualified-Variables \) "," \( Implicitly-Qualified-Variables \)
\langle Extended subexpression implementation 1570 \rangle + \equiv
procedure extSubexpObj.FinishExistential;
  var k: integer;
  begin for k \leftarrow nQualifiedSegments.Count - 1 downto 1 do { from inside outwards }
     begin qLastFormula \leftarrow new(ExistentialFormulaPtr,
           Init(QualifiedSegmentPtr(nQualifiedSegments.Items \uparrow [k]) \uparrow .nSegmPos,
           nQualifiedSegments.Items \uparrow [k], gLastFormula)); nQualifiedSegments.Items \uparrow [k] \leftarrow \mathbf{nil};
     end;
  if nQualifiedSegments.Count > 0 then
     \textbf{begin} \ gLastFormula \leftarrow new(ExistentialFormulaPtr, Init(nSubexpPos, nQualifiedSegments.Items \uparrow [0],
           gLastFormula)); nQualifiedSegments.Items \uparrow [0] \leftarrow nil;
     end:
  nQualifiedSegments.Done;
  end;
1668.
          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.
\langle Extended subexpression implementation 1570 \rangle + \equiv
procedure extSubexpObj.FinishUniversal;
  var k: integer;
  begin if nRestriction \neq nil then { transform st into implies }
     qLastFormula \leftarrow new(ConditionalFormulaPtr, Init(nRestrPos, nRestriction, qLastFormula));
  for k \leftarrow nQualifiedSegments.Count - 1 downto 1 do
     begin qLastFormula \leftarrow new(UniversalFormulaPtr,
           Init(QualifiedSegmentPtr(nQualifiedSegments.Items \uparrow [k]) \uparrow .nSegmPos,
           nQualifiedSegments.Items \uparrow [k], gLastFormula)); nQualifiedSegments.Items \uparrow [k] \leftarrow \mathbf{nil};
     end;
  if nQualifiedSegments.Count > 0 then
     begin qLastFormula \leftarrow new(UniversalFormulaPtr, Init(nSubexpPos, nQualifiedSegments.Items \uparrow [0],
           gLastFormula)); nQualifiedSegments.Items \uparrow [0] \leftarrow nil;
     end;
```

end;

Section 23.4. EXTENDED EXPRESSION CLASS

begin $gSubexpPtr \leftarrow new(extSubexpPtr, Init)$

end;

1669. When an expression is needed, the qExpPtr 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 1669 $\rangle \equiv$ $extExpressionPtr = \uparrow extExpressionObj;$ extExpressionObj = object (ExpressionObj)**constructor** *Init*(*fExpKind* : *ExpKind*); procedure CreateSubexpression; virtual; end; This code is used in section 1379. 1670. Constructor. This just invokes the parent class's constructor (§871), then resets the module-wide variable TermNbr to zero. \langle Extended expression implementation $1670 \rangle \equiv$ **constructor** *extExpressionObj.Init(fExpKind : ExpKind)*; **begin** $inherited Init(fExpKind); TermNbr \leftarrow 0;$ end; See also section 1671. This code is used in section 1380. An extExpression creating a subexpression overrides the parent class's method ($\S872$), and sets the global gSubexpPtr to point to a new extSubexp object. \langle Extended expression implementation $1670 \rangle + \equiv$ **procedure** *extExpressionObj.CreateSubexpression*;

524 PARSER Mizar Parser $\S1672$

File 24

Parser

1672. The Parser has a "big red button": a single "obvious" function for the user to, you know, push. Namely, the *Parse* procedure ($\S\S1921\ et\ seq.$). 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 1672⟩ ≡
⟨ GNU License 4⟩
unit parser;
interface
uses mscanner;
var StillCorrect: boolean = true;
type ReadTokenProcedure = Procedure;
const ReadTokenProc: ReadTokenProcedure = ReadToken; {from mscanner.pas}
procedure Parse;
procedure SemErr(fErrNr : integer);
implementation
uses syntax, errhan, pragmas
    mdebug , info end_mdebug;
⟨ Implementation of parser.pas 1673⟩
```

1673. 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 1673⟩ ≡
  ⟨ Local constants for parser.pas 1674⟩;
  ⟨ Parse expressions (parser.pas) 1687⟩
  ⟨ Communicate with items (parser.pas) 1773⟩
  ⟨ Process miscellany (parser.pas) 1774⟩
  ⟨ Parse simple justifications (parser.pas) 1791⟩
  ⟨ Parse statements and reasoning (parser.pas) 1802⟩
  ⟨ Parse patterns (parser.pas) 1835⟩
  ⟨ Parse definitions (parser.pas) 1868⟩
  ⟨ Parse scheme block (parser.pas) 1915⟩
  ⟨ Main parse method (parser.pas) 1921⟩
See also sections 1676, 1677, 1679, 1680, 1682, 1683, and 1684.
This code is used in section 1672.
```

 $\S1674$ Mizar Parser PARSER 525

1674. We have error codes for syntactically invalid situations. These are all different ways for panic to occur (hence the "pa-" prefix). We will introduce the error codes when they are first used. The unused error codes are listed below.

```
\langle \text{Local constants for parser.pas 1674} \rangle \equiv
const (Error codes for parser 1675)
See also section 1678.
This code is used in section 1673.
        \langle \text{Error codes for parser 1675} \rangle \equiv
  paUnexpAntonym1 = 198; paUnexpAntonym2 = 198; paUnexpSynonym = 199; paUnexpHereby = 216;
  paUnexpReconsider = 228; \ paIdentExp5 = 300; \ paIdentExp12 = 300; \ paWrongRightBracket1 = 311;
  paWrongRightBracket2 = 311; paWrongPattBeg3 = 314; paRightSquareExp1 = 371;
  paRightSquareExp3 = 371; paRightCurledExp2 = 372; paWronqAttrPrefixExpr = 375;
  paWrongAttrArgumentSuffix = 376; paTypeExpInAdjectiveCluster = 377;
  paTypeUnexpInClusterRegistration = 405;
See also sections 1681, 1694, 1697, 1699, 1701, 1704, 1707, 1710, 1712, 1715, 1717, 1723, 1726, 1728, 1730, 1732, 1741, 1743,
    1745, 1750, 1753, 1755, 1759, 1761, 1763, 1765, 1771, 1784, 1786, 1788, 1793, 1795, 1797, 1799, 1801, 1806, 1808, 1810,
    1812, 1814, 1824, 1826, 1831, 1836, 1839, 1841, 1844, 1846, 1848, 1850, 1852, 1854, 1857, 1860, 1862, 1865, 1867, 1874,
    1876, 1878, 1882, 1884, 1886, 1890, 1896, 1899, 1901, 1905, 1907, 1909, 1912, 1916, and 1918.
This code is used in section 1674.
1676. (Implementation of parser.pas 1673) +\equiv
var qAddSymbolsSet: set of char = []; { not used anywhere }
1677. Syntax errors do three things:
(1) Marks StillCorrect to be false (i.e., enters panic mode)
(2) Reports the error with the ErrImm (§130) 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 1673\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.
  Note: qMainSet is only used in the SynErr procedure, and nowhere else in Mizar.
\langle Local constants for parser.pas 1674 \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];
```

526 PARSER Mizar Parser $\S1679$

We have a few more methods for *specific situations* where errors are likely to occur.

```
\langle Implementation of parser.pas 1673 \rangle + \equiv
procedure MissingWord(fErrNr: integer);
  var lPos: Position;
  begin lPos \leftarrow PrevPos; inc(lPos.Col); SynErr(lPos,fErrNr)
procedure WrongWord(fErrNr: integer);
  begin SynErr(CurPos, fErrNr)
  end:
1680.
        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
(8846) mutates the global state.
  The Semicolon procedure should probably match the AcceptEnd procedure — i.e., it should be of the
form "if \langle Current \ token \ is \ semicolon \rangle then ReadTokenProc \ else \ \langle Flag \ error \rangle".
\langle Implementation of parser.pas 1673 \rangle + \equiv
procedure Semicolon;
  begin KillItem;
  if CurWord.Kind \neq sy\_Semicolon then MissingWord(paSemicolonExp);
  if CurWord.Kind = sy\_Semicolon then ReadTokenProc;
procedure AcceptEnd(fPos : Position);
  begin if CurWord.Kind = sy\_End then ReadTokenProc
  else begin Error (fPos, paEndExp); MissingWord (paUnpairedSymbol)
    end;
  end;
        \langle \text{Error codes for parser 1675} \rangle + \equiv
  paUnpairedSymbol = 214; paEndExp = 215; paSemicolonExp = 330;
1682. Due to the structure of PASCAL, the Parser will frequently be in situations where we consider the
case of the current kind of token, and for "valid" branches we will want the Parser to consume the current
token and move on. For example, if the Parser is looking at an open bracket.
  But if the Parser is a panicking mess, then we should raise an error to alert the user.
  Either some explanation should be offered for the magic number 2546 = #9f2, or it should be stored in
a constant (or a WEB macro).
\langle Implementation of parser.pas 1673\rangle + \equiv
procedure ReadWord;
  begin Mizassert (2546, StillCorrect); ReadTokenProc
  end;
```

 $\S1683$ Mizar Parser PARSER 527

1683. 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.

When will an error be flagged? If the Parser is panicking, or if the current token does not match the expected token.

```
⟨ Implementation of parser.pas 1673⟩ +≡
function Occurs(fW : TokenKind): boolean;
begin Occurs ← false;
if CurWord.Kind = fW then
begin ReadWord; Occurs ← true end
end;
procedure Accept(fCh : TokenKind; fErrNr : integer);
begin if ¬Occurs(fCh) then MissingWord(fErrNr)
end;
```

1684. 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 1673⟩ +≡
procedure SemErr(fErrNr: integer);
begin if StillCorrect then ErrImm(fErrNr)
end;
```

1685. Exercise: For each procedure and function we are about to define in the rest of the Parser, when will an error be raised and by which of these functions?

528 EXPRESSIONS Mizar Parser $\S1686$

Section 24.1. EXPRESSIONS

1686. The syntactic classes we're interested in (terms, types, formulas) almost always appear as subexpressions in a formula or some other expression. The Parser works with various procedures to parse these guys as subexpressions: TermSubexpression (§1720), TypeSubexpression (§1735), FormulaSubexpression (§1772). When we need a term (or type or formula) as an expression, as we will in the next section, we use these procedures to construct the abstract syntax tree.

Subsection 24.1.1. Terms

1687. 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,

This code is used in section 1673.

- (7) placeholder variables appearing in private functors and predicates,
- (8) structure symbols.

```
 \langle \, \text{Parse expressions (parser.pas)} \, \, 1687 \rangle \equiv \\ \{ \, \textit{Expressions} \, \} \\ \text{const } \, \textit{TermBegSys: set of} \\ \text{TokenKind} = [ \, \textit{Identifier}, \, \textit{InfixOperatorSymbol}, \, \textit{Numeral}, \, \textit{LeftCircumfixSymbol}, \, \textit{sy\_LeftParanthesis}, \\ \text{sy\_It}, \, \textit{sy\_LeftCurlyBracket}, \, \textit{sy\_LeftSquareBracket}, \, \textit{sy\_The}, \, \textit{sy\_Dolar}, \, \textit{Structuresymbol} ]; \\ \text{See also sections 1688, 1689, 1690, 1691, 1695, 1713, 1714, 1716, 1724, 1727, 1735, 1736, 1737, 1738, 1739, 1742, 1744, 1746, \\ 1747, \, 1751, \, 1752, \, 1760, \, 1766, \, 1767, \, 1768, \, 1769, \, 1770, \, \text{and } 1772. \\ \end{cases}
```

1688. We have a few helper function for Accept-ing parentheses. This invokes the ProcessLeftParenthesis method for the gSubexpPtr (§850) global variable which we recall (§878) 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.

[The ClosedParenth method should raise an error when the user passes a negative value for fParenthCnt, but that may be "too defensive".]

```
 \langle \operatorname{Parse\ expressions\ }(\operatorname{parser\ .pas})\ 1687 \rangle + \equiv \\  \operatorname{procedure\ } \operatorname{OpenParenth\ }(\operatorname{var\ } \operatorname{FParenth\ } \operatorname{Cnt}: \operatorname{integer}); \\  \operatorname{begin\ } \operatorname{FParenth\ } \operatorname{Cnt} \leftarrow 0; \\  \operatorname{while\ } \operatorname{Cur\ } \operatorname{Word\ .} \operatorname{Kind\ } = \operatorname{sy\ .} \operatorname{LeftParanthesis\ } \operatorname{do\ } \\  \operatorname{begin\ } \operatorname{gSubexp\ } \operatorname{Ptr\ } \cap \operatorname{Process\ } \operatorname{LeftParenth\ } \operatorname{end\ }; \\  \operatorname{end\ }; \\  \operatorname{end\ }; \\  \operatorname{end\ }; \\  \operatorname{fParenth\ } \operatorname{Cnt\ } \geq 0 \ \} \\  \operatorname{end\ }; \\  \operatorname{procedure\ } \operatorname{CloseParenth\ }(\operatorname{var\ } \operatorname{FParenth\ } \operatorname{Cnt\ } : \operatorname{integer\ }); \\  \operatorname{begin\ } \operatorname{while\ } (\operatorname{Cur\ } \operatorname{Word\ .} \operatorname{Kind\ } = \operatorname{sy\ .} \operatorname{Right\ } \operatorname{Paranth\ } \operatorname{ent\ } \wedge \operatorname{int\ } \cap \operatorname{In
```

 $\S1689$ Mizar Parser TERMS 529

1689. 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 (§878). So what's going on?

Well, the only work being done here is in the branch handling "qua", specifically the gSubexpPtr state variable marks the "qua" position (§1581), the next word is read, and then control is handed off to the Parser's TypeSubexpression procedure. The AST is assembled with the FinishQualifiedTerm (§1582) method.

```
⟨ Parse expressions (parser.pas) 1687⟩ +≡
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;
end;
```

1690. Parsing the contents of a bracketed term starts a bracketed term ($\S1622$), reads the next word after the start of the bracket, then consumes the maximum number of visible arguments ($\S842$). The gSubexpPtr constructs the AST for the bracketed term and its contents ($\S1623$).

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 \, {\it Parse expressions (parser.pas)} \, \, 1687 \rangle + \equiv {\it procedure GetArguments (const fArgsNbr: integer); forward;} \ {\it procedure BracketedTerm;} \ {\it begin gSubexpPtr} \uparrow . StartBracketedTerm; ReadWord; GetArguments (MaxVisArgNbr); gSubexpPtr \uparrow . FinishBracketedTerm; end;} \ {\it end;} \
```

1691. 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) 1687⟩ +≡ procedure TermSubexpression; forward; procedure FormulaSubexpression; forward; procedure ArgumentsTail(fArgsNbr:integer); forward; procedure ProcessPostqualification; begin gSubexpPtr\uparrow.StartPostqualification; {(§1612)} while CurWord.Kind = sy\_Where do begin repeat ⟨ Process post-qualified segment 1692⟩ until CurWord.Kind \neq sy\_Comma; end; end;
```

530 TERMS Mizar Parser $\S1692$

1692. Each "segment" in a post-qualification looks like:

```
\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.

```
define parse_post_qualified_type ≡ begin ReadWord; TypeSubexpression; end

⟨Process post-qualified segment 1692⟩ ≡
gSubexpPtr↑.StartPostQualifyingSegment; {(§1613)} ReadWord;
⟨Parse post-qualified comma-separated list of variables 1693⟩;
gSubexpPtr↑.StartPostqualificationSpecyfication; {(§1615)}
if CurWord.Kind ∈ [sy_Is, sy_are] then parse_post_qualified_type;
gSubexpPtr↑.FinishPostqualifyingSegment; {(§1616)}

This code is used in section 1691.

1693. ⟨Parse post-qualified comma-separated list of variables 1693⟩ ≡
repeat gSubexpPtr↑.ProcessPostqualifiedVariable; {(§1614)} Accept(Identifier, paIdentExp1);
until ¬Occurs(sy_Comma)

This code is used in section 1692.

1694. ⟨Error codes for parser 1675⟩ +≡
paIdentExp1 = 300; paRightParenthExp1 = 370;
```

1695. 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):

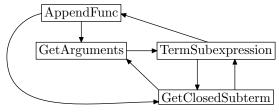


Fig. 14. Control flow when parsing a term.

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) 1687⟩ +≡

procedure GetClosedSubterm;

begin case CurWord.Kind of

⟨Get closed subterm of identifier 1696⟩;

⟨Get closed subterm of structure 1698⟩;

Numeral: begin gSubexpPtr↑.ProcessNumeralTerm; ReadWord end;

⟨Get closed subterm of bracketed expression 1700⟩;

sy_It: begin gSubexpPtr↑.ProcessItTerm; ReadWord end;

sy_Dolar: begin gSubexpPtr↑.ProcessLocusTerm; ReadWord end;

⟨Get closed subterm of Fraenkel operator or enumerated set 1702⟩;

⟨Get closed subterm of choice operator 1706⟩;

othercases RunTimeError(2133);
endcases;
end;
```

 $\{1696$ Mizar Parser TERMS 531

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. Remember, private functors which omit the closing right parentheses should be flagged with a 370 error. \langle Get closed subterm of identifier $1696 \rangle \equiv$ Identifier: **if** AheadWord.Kind = $sy_LeftParanthesis$ **then** { treat identifier as private functor } **begin** *qSubexpPtr*↑.*StartPrivateTerm*; *ReadWord*; *ReadWord*; if $CurWord.Kind \neq sy_RightParanthesis$ then GetArguments(MaxVisArgNbr); $qSubexpPtr \uparrow .FinishPrivateTerm; Accept(sy_RightParanthesis, paRightParenthExp2);$ end { treat identifier as variable } **begin** $gSubexpPtr\uparrow.ProcessSimpleTerm; \{(\S1580)\}\ ReadWord$ This code is used in section 1695. **1697.** $\langle \text{Error codes for parser } 1675 \rangle + \equiv$ paRightParenthExp2 = 370;1698. 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. A 363 error will be raised if the "(#" is missing, and a 373 error will be raised if the "#)" structure bracket is missing. \langle Get closed subterm of structure $1698 \rangle \equiv$ StructureSymbol: **begin** $gSubexpPtr \uparrow .StartAggregateTerm; ReadWord;$ $Accept(sy_StructLeftBracket, paLeftDoubleExp1); GetArguments(MaxVisArgNbr);$ $qSubexpPtr\uparrow.FinishAggregateTerm; \{(\S1625)\}\ Accept(sy_StructRightBracket,paRightDoubleExp1);$ end This code is used in section 1695. **1699.** $\langle \text{Error codes for parser } 1675 \rangle + \equiv$ paLeftDoubleExp1 = 363; paRightDoubleExp1 = 373; 1700. Encountering a left bracket of some kind — specifically a user-defined left bracket or a "[" — should cause the Parser to look for the contents of a bracketed term (§1690), then a right bracket. \langle Get closed subterm of bracketed expression $1700 \rangle \equiv$ $LeftCircumfixSymbol, sy_LeftSquareBracket$: **begin** BracketedTerm; case Curword.Kind of $sy_RightSquareBracket$, $sy_RightCurlyBracket$, $sy_RightParanthesis$: ReadWord; **othercases** Accept(RightCircumfixSymbol, paRightBraExp1); endcases: end This code is used in section 1695. 1701. $\langle \text{Error codes for parser } 1675 \rangle + \equiv$ paRightBraExp1 = 310;

532 TERMS Mizar Parser $\S1702$

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 1702 $\rangle \equiv$ $sy_LeftCurlyBracket$: begin $qSubexpPtr\uparrow.StartBracketedTerm$; $\{(\S1622)\}$ ReadWord; TermSubexpression; { (§1720) } if $(CurWord.Kind = sy_Colon) \lor (CurWord.Kind = sy_Where)$ then $\langle Parse \text{ a Fraenkel operator } 1703 \rangle$ else $\langle Parse an enumerated set 1705 \rangle$; end This code is used in section 1695. 1703. Parsing a Fraenkel operator, well, we recall Fraenkel operators look like $\{\langle term \rangle \langle post\text{-}qualified \ segment \rangle \ ":" \ \langle formula \rangle \}$ $\langle \text{ Parse a Fraenkel operator } 1703 \rangle \equiv$ **begin** $qSubexpPtr\uparrow.StartFraenkelTerm; ProcessPostqualification; <math>qSubexpPtr\uparrow.FinishSample;$ $Accept(sy_Colon, paColonExp1); FormulaSubexpression; gSubexpPtr \uparrow. FinishFraenkelTerm;$ $Accept(sy_RightCurlyBracket, paRightCurledExp1);$ end This code is used in section 1702. 1704. $\langle \text{Error codes for parser } 1675 \rangle + \equiv$ paRightCurledExp1 = 372; paColonExp1 = 384;The Parser can also run into a finite set $\{x_1, \ldots, x_n\}$. The braces are treated like any other functor bracket, in the sense that if the right brace \} is missing, then a 310 error will be raised. $\langle \text{ Parse an enumerated set } 1705 \rangle \equiv$ **begin** $gSubexpPtr\uparrow.FinishArgument; ArgumentsTail(MaxVisArgNbr - 1);$ $gSubexpPtr \uparrow . FinishBracketedTerm;$ case Curword.Kind of $sy_RightSquareBracket$, $sy_RightCurlyBracket$, $sy_RightParanthesis$: ReadWord; **othercases** Accept(RightCircumfixSymbol, paRightBraExp1); endcases;

This code is used in section 1702.

end

 $\{1706 \quad \text{Mizar Parser}$ TERMS 533

1706. Mizar allows "the" to be used for selector functors, forgetful functors, choice operators, or simple Fraenkel terms.

Note we are generous *here* with what situations leads to treating "the" as a choice operator, because in other parsing procedures any mistakes will be caught there.

```
define choice\_operator\_cases \equiv ModeSymbol, AttributeSymbol, sy\_Non, sy\_LeftParanthesis, Identifier,
                InfixOperatorSymbol, Numeral, LeftCircumfixSymbol, sy_{-}It, sy_{-}LeftCurlyBracket,
               sy\_LeftSquareBracket, sy\_The, sy\_Dolar
\langle Get closed subterm of choice operator 1706 \rangle \equiv
sy\_The: begin gSubexpPtr \uparrow .ProcessThe; ReadWord;
  case CurWord.Kind of
  SelectorSymbol: \langle Parse selector functor 1708 \rangle;
  StructureSymbol: (Parse forgetful functor or choice of structure type 1709);
  sy\_Set: \langle Parse simple Fraenkel expression or "the set" 1711 <math>\rangle;
  choice\_operator\_cases: begin qSubexpPtr \uparrow .StartChoiceTerm; TypeSubexpression;
     gSubexpPtr \uparrow. FinishChoiceTerm;
     end
  othercases begin gSubexpPtr\uparrow.InsertIncorrTerm;\ WrongWord(paWrongAfterThe)
     end;
  endcases;
  end
This code is used in section 1695.
1707. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paWrongAfterThe = 320;
1708. \langle \text{Parse selector functor } 1708 \rangle \equiv
  begin gSubexpPtr\uparrow.StartSelectorTerm; ReadWord; { parses "the <math>\langle selector \rangle" }
  if Occurs(sy\_Of) then TermSubexpression; { parses "of \langle Term \rangle" }
  gSubexpPtr\uparrow.FinishSelectorTerm; { builds AST subtree }
This code is used in section 1706.
1709. 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 1709\rangle \equiv
  if AheadWord.Kind = sy\_Of then { forgetful functor }
     begin qSubexpPtr\uparrow.StartForgetfulTerm; ReadWord; Accept(sy_Of, paOfExp); TermSubexpression;
     qSubexpPtr\uparrow.FinishForgetfulTerm;
     end
          { choice operator, e.g., "the multMagma" }
  begin gSubexpPtr\uparrow.StartChoiceTerm; TypeSubexpression; <math>gSubexpPtr\uparrow.FinishChoiceTerm;
  end
This code is used in section 1706.
1710. \langle \text{Error codes for parser } 1675 \rangle + \equiv
```

paOfExp = 256;

534 TERMS Mizar Parser $\S1711$

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. \langle Parse simple Fraenkel expression or "the set" 1711 $\rangle \equiv$ if $AheadWord.Kind = sy_Of$ then { simple Fraenkel expression } **begin** ReadWord; $\{ set \}$ ReadWord; { of } $gSubexpPtr\uparrow.StartSimpleFraenkelTerm;\ Accept(sy_All, paAllExp);\ TermSubexpression;$ $gSubexpPtr\uparrow.StartFraenkelTerm;\ ProcessPostqualification;\ gSubexpPtr\uparrow.FinishSimpleFraenkelTerm;$ end { "the set" } **begin** $qSubexpPtr\uparrow.StartChoiceTerm; TypeSubexpression; <math>gSubexpPtr\uparrow.FinishChoiceTerm;$ **end** This code is used in section 1706. $\langle \text{Error codes for parser } 1675 \rangle + \equiv$ 1712. paAllExp = 275;1713. Subexpression object's FinishArgument (§1585) is invoked, which pushes a term onto the Term stack. This will invoke the AppendQua (§1689) method and expect a closed parentheses afterwards (§1688). Possible bug: what should happen when fParenthCnt is zero or negative? $\langle \text{Parse expressions (parser.pas) } 1687 \rangle + \equiv$ procedure CompleteArgument(var fParenthCnt : integer); **begin** $gSubexpPtr\uparrow.FinishArgument;$ **repeat** AppendQua; CloseParenth(fParenthCnt);**until** $CurWord.Kind \neq sy_Qua; \{ \land (CurWord.Kind \neq sy_Exactly) \}$

end;

 $\S1714$ Mizar Parser TERMS 535

1714. 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 (§850) was a global variable declared back in syntax.pas, and the CreateSubexpression (§1671) mutates the gSubexpPtr variable. Now we see it in action.

This invokes the ProcessLeftParenthesis method for the gSubexpPtr (§850) global variable which we recall (§878) is an empty virtual method. So the Parser just "consumes" a left parentheses.

Note that the **case** expression considers the type of TokenKind (§884) 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) } 1687 \rangle + \equiv
procedure AppendFunc(var fParenthCnt: integer);
  begin while CurWord.Kind = InfixOperatorSymbol do
    begin gSubexpPtr\uparrow.StartLongTerm; { (§1593) }
    repeat qSubexpPtr\uparrow.ProcessFunctorSymbol; { <math>(\S1609) }
       ReadWord:
       case CurWord.Kind of
       sy\_LeftParanthesis:
         begin
                   { parenthetised term(s) }
         gSubexpPtr\uparrow.ProcessLeftParenthesis; ReadWord; {consume the left paren}
         GetArguments(MaxVisArgNbr); \{ (\S1738) \}
         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; { (§1671) }
         GetClosedSubterm; \{ (\S 1695) \}
         qSubexpPtr\uparrow.FinishArgument; \{ (\S1585) \}
         KillSubexpression; \{ (\S 844) \}
         end:
       endcases;
       gSubexpPtr\uparrow.FinishArgList; \{ (\S 1610) \}
    until CurWord.Kind \neq InfixOperatorSymbol;
    gSubexpPtr\uparrow.FinishLongTerm; \{ (\S1597) \}
    CompleteArgument(fParenthCnt); \{ (\S1713) \}
    end;
  end;
       \langle \text{Error codes for parser 1675} \rangle + \equiv
  paRightParenthExp3 = 370;
```

536 TERMS Mizar Parser $\S1716$

```
1716. 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) } 1687 \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 1718);
  (Check every remaining open (left) parentheses has a corresponding partner 1719);
  end:
1717. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paWrongTermBeg = 397;
1718. (Keep parsing as long as there is an infixed operator to the right 1718) \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 1716.
1719. (Check every remaining open (left) parentheses has a corresponding partner 1719) \equiv
  while lParenthCnt > 0 do
    begin gSubexpPtr\uparrow.ProcessRightParenthesis; Accept(sy_RightParenthesis, paRightParenthExp1);
    dec(lParenthCnt);
    end
This code is used in section 1716.
```

 $\S1720$ Mizar Parser TERMS 537

1720. Term subexpressions. The Parser wants a term as a subexpression in a formula or attribute cluster or some similar situation. The term specifically is just a *component* of the expression. We should recall from Figure 14 ($\S1695$) that this is a critical part of parsing terms.

```
\langle \text{Parse term subexpressions (parser.pas) } 1720 \rangle \equiv
procedure TermSubexpression;
  var lParenthCnt: integer;
  begin gExpPtr\uparrow.CreateSubexpression; OpenParenth(lParenthCnt); { (§1688) }
  case CurWord.Kind of
  Identifier, Numeral, LeftCircumfixSymbol, sy\_It, sy\_LeftCurlyBracket, sy\_LeftSquareBracket, sy\_The,
         sy\_Dolar, StructureSymbol:
    begin GetClosedSubterm; CompleteArgument(lParenthCnt); \{(\S1713)\}
    end:
  InfixOperatorSymbol:
                            \{ skip \} ;
  othercases (Raise error over invalid term subexpression 1721);
  endcases;
  AppendFunc(lParenthCnt); \{ (\S1714) \}
  while lParenthCnt > 0 do \langle Parse arguments to the right 1722 \rangle;
  gSubexpPtr \uparrow. FinishTerm; KillSubexpression;
  end;
This code is used in section 1736.
1721. \langle Raise error over invalid term subexpression 1721 \rangle \equiv
  begin gSubexpPtr\uparrow.InsertIncorrTerm; gSubexpPtr\uparrow.FinishArgument; WrongWord(paWrongTermBeg);
  end
This code is used in section 1720.
1722. \langle \text{ Parse arguments to the right } 1722 \rangle \equiv
  begin ArgumentsTail(MaxVisArgNbr-1); dec(lParenthCnt); gSubexpPtr \uparrow. ProcessRightParenthesis;
  Accept(sy\_RightParanthesis, paRightParenthExp10);
  if CurWord.Kind \neq InfixOperatorSymbol then MissingWord(paFunctExp3);
  AppendFunc(lParenthCnt);
  end
This code is used in section 1720.
         \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paFunctExp3 = 302; paRightParenthExp10 = 370;
```

Subsection 24.1.2. Types and Attributes

1724. Types and attributes are closely related, when it comes to parsing Mizar. After all, we can add an adjective to a type and we expect it to be "a type".

An adjective cluster is just one or more (possibly negated) attribute.

```
⟨ Parse expressions (parser.pas) 1687⟩ +≡
⟨ Process attributes (parser.pas) 1725⟩
procedure GetAdjectiveCluster;
begin gSubexpPtr↑.StartAdjectiveCluster; ProcessAttributes; gSubexpPtr↑.FinishAdjectiveCluster;
end;
```

538 TYPES AND ATTRIBUTES Mizar Parser §1725

1725.

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) 1725 \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;
     if ahead_is_attribute_argument then
       begin gSubexpPtr \uparrow. StartAttributeArguments; ProcessArguments;
       gSubexpPtr \uparrow. CompleteAttributeArguments;
       end;
    if CurWord.Kind = AttributeSymbol then
       begin gSubexpPtr\uparrow.ProcessAttribute; ReadWord; end
     else begin SynErr(CurPos, paAttrExp1)
       end;
     end;
  end;
This code is used in section 1724.
1726. \langle Error codes for parser 1675 \rangle + \equiv
  paAttrExp1 = 306;
1727. 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) } 1687 \rangle + \equiv
\mathbf{procedure} \ \mathit{RadixTypeSubexpression};
  var lSymbol, lParenthCnt: integer;
  begin lParenthCnt \leftarrow 0; \langle Parse optional left-paren 1733 \rangle;
  qSubexpPtr\uparrow.ProcessModeSymbol; \{(\S1588)\}
  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 1729 \rangle;
  StructureSymbol: \langle Parse structure as radix type 1731 \rangle;
  othercases begin MissingWord(paWrongRadTypeBeg); gSubexpPtr\uparrow.InsertIncorrType end;
  endcases:
  \langle Close the parentheses 1734\rangle;
  gSubexpPtr \uparrow . FinishType;
  end;
1728. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paWrongRadTypeBeg = 398;
```

```
1729.
         \langle \text{ Parse mode as radix type } 1729 \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 1727.
1730. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paUnexpOf = 183;
1731. \langle \text{ Parse structure as radix type } 1731 \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 1727.
1732. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paUnexpOver = 184;
1733. \langle \text{Parse optional left-paren } 1733 \rangle \equiv
  if CurWord.Kind = sy\_LeftParanthesis then
     begin gSubexpPtr\uparrow.ProcessLeftParenthesis; ReadWord; inc(lParenthCnt);
This code is used in section 1727.
1734. \langle Close the parentheses 1734\rangle \equiv
  if lParenthCnt > 0 then
     begin gSubexpPtr\uparrow.ProcessRightParenthesis; Accept(sy_RightParenthesis, paRightParenthExp1);
     end
This code is used in section 1727.
         Type subexpressions. Now the Parser needs a type as a subexpression in a larger expression
(e.g., the specification for a definition, or in a formula of the form "\langle Term \rangle is \langle Type \rangle"). We basically get
the adjectives with GetAdjectiveCluster, then we get the radix type with RadixTypeSubexpression.
\langle \text{Parse expressions (parser.pas) } 1687 \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;
        Aside: parsing term subexpressions. The code for parsing term subexpressions (§1720) appears
```

here in the code for the Parser, but it felt out of place. I thought it best to place it at the end of the subsection on parsing Term expressions (as it is the pinnacle of Term parsing), rather than leave it here. ⟨ Parse expressions (parser.pas) 1687⟩ += ⟨ Parse term subexpressions (parser.pas) 1720⟩ 1737. 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. $\langle Parse \ expressions \ (parser.pas) \ 1687 \rangle +\equiv$

```
⟨ Parse expressions (parser.pas) 1687⟩ +≡
procedure ArgumentsTail(fArgsNbr: integer);
begin while (fArgsNbr > 0) ∧ Occurs(sy_Comma) do
    begin gSubexpPtr↑.StartArgument; TermSubexpression; gSubexpPtr↑.FinishArgument;
    dec(fArgsNbr);
    end;
end;
end;

1738. Attributes, terms, predicates have terms as arguments. This relies upon the FinishArguments
method (§1585).
⟨ Parse expressions (parser.pas) 1687⟩ +≡
procedure GetArguments(const fArgsNbr: integer);
begin if fArgsNbr > 0 then
    begin TermSubexpression; gSubexpPtr↑.FinishArgument; ArgumentsTail(fArgsNbr - 1);
    end;
```

Subsection 24.1.3. Formulas

end;

1739. 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) } 1687 \rangle +\equiv \\ \textbf{procedure } \textit{QuantifiedVariables}; \\ \textbf{begin repeat } \textit{gSubexpPtr} \uparrow. \textit{StartQualifiedSegment}; \textit{ReadWord}; \\ \langle \text{ Parse comma-separated variables for quantified variables } 1740 \rangle; \\ \textit{gSubexpPtr} \uparrow. \textit{StartQualifyingType}; \\ \textbf{if } \textit{Occurs}(\textit{sy\_Be}) \lor \textit{Occurs}(\textit{sy\_Being}) \textbf{ then } \textit{TypeSubexpression}; \\ \textit{gSubexpPtr} \uparrow. \textit{FinishQualifiedSegment}; \\ \textbf{until } \textit{CurWord.Kind} \neq \textit{sy\_Comma}; \\ \textbf{end}; \\ \end{aligned}
```

1740. ⟨Parse comma-separated variables for quantified variables 1740⟩ ≡ **repeat** gSubexpPtr↑.ProcessVariable; Accept(Identifier, paIdentExp2); **until** ¬Occurs(sy_Comma)

This code is used in section 1739.

```
1741. \langle \text{Error codes for parser } 1675 \rangle + \equiv paIdentExp2 = 300;
```

 $\S1742$ Mizar Parser FORMULAS 541

1742. The existential formula looks like

```
ex \langle Quantified\text{-}Variables \rangle \text{ st } \langle Formula \rangle
```

The Parser implements it quite faithfully. $\langle \text{Parse expressions (parser.pas) } 1687 \rangle + \equiv$ **procedure** ExistentialFormula; **begin** gSubexpPtr \uparrow .StartExistential; QuantifiedVariables;
gSubexpPtr \uparrow .FinishQuantified; Accept(sy_St, paStExp); FormulaSubexpression;
gSubexpPtr \uparrow .FinishExistential;
end:

1743. $\langle \text{Error codes for parser } 1675 \rangle + \equiv paStExp = 387;$

1744. 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.

```
\langle Parse expressions (parser.pas) 1687\rangle +\equiv procedure UniversalFormula;
```

begin $gSubexpPtr\uparrow.StartUniversal;$ QuantifiedVariables; $gSubexpPtr\uparrow.FinishQuantified;$ **if** $CurWord.Kind = sy_St$ **then**

begin $gSubexpPtr\uparrow.StartRestriction;$ ReadWord; FormulaSubexpression; $gSubexpPtr\uparrow.FinishRestriction;$ **end**:

case CurWord.Kind of

 $sy_Holds: \ \mathbf{begin} \ gSubexpPtr \uparrow. ProcessHolds; \ ReadWord \ \mathbf{end};$

 sy_For, sy_Ex : ; {fallthrough} othercases begin $gSubexpPtr \uparrow .InsertIncorrFormula$; MissingWord(paWrongScopeBeg)

end; endcases;

FormulaSubexpression; $gSubexpPtr\uparrow.FinishUniversal$; end;

1745. $\langle \text{Error codes for parser } 1675 \rangle + \equiv paWrongScopeBeg = 340;$

542 FORMULAS Mizar Parser $\S1746$

1746. 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 (§1687) which is all the token kinds for starting a term. If the next token is a term, then GetArguments is invoked to parse them.

```
 \begin gSubexpPtr \uparrow. FinishRightSideOfPredicativeFormula; \\ end; \\ end; \\ \begin gSubexpPtr \uparrow. ProcessRightSideOfPredicateSymbol; ReadWord; \\ \begin gSubexpPtr \uparrow. ProcessRightSideOfPredicateSymbol; ReadWord; \\ \begin gSubexpPtr \uparrow. FinishRightSideOfPredicativeFormula; \\ \begin gSubexpPtr \uparrow. FinishRightS
```

1747. 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) } 1687 \rangle + \equiv
procedure CompleteMultiPredicativeFormula;
  begin gSubexpPtr\uparrow.StartMultiPredicativeFormula;
  repeat case CurWord.Kind of
    sy\_Equal, PredicateSymbol: CompleteRightSideOfThePredicativeFormula(CurWord.Nr);
    sy_Does, sy_Do: \(\rangle\) Parse multi-predicate with "does" or "do" in copula 1748\);
    endcases;
  until \neg (CurWord.Kind \in [sy\_Equal, PredicateSymbol, sy\_Does, sy\_Do]);
  gSubexpPtr \uparrow . FinishMultiPredicativeFormula;
  end:
1748. \langle \text{Parse multi-predicate with "does" or "do" in copula 1748} \rangle \equiv
  begin (Consume "does not" or "do not", raise error otherwise 1749);
  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 1747.
1749. (Consume "does not" or "do not", raise error otherwise 1749) \equiv
  qSubexpPtr\uparrow.ProcessDoesNot;\ ReadWord;\ Accept(sy\_Not,paNotExpected)
This code is used in section 1748.
```

§1750 FORMULAS 543 Mizar Parser

```
1750.
         \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paNotExpected = 401; paInfinitiveExp = 402;
1751. 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.
\langle \text{Parse expressions (parser.pas) } 1687 \rangle + \equiv
procedure CompletePredicativeFormula(aPredSymbol: integer);
  begin qSubexpPtr\uparrow.ProcessPredicateSymbol; { (§1640) }
  ReadWord:
  if CurWord.Kind \in TermBegSys then GetArguments(PredMaxArgs.fList \uparrow [aPredSymbol]);
  gSubexpPtr\uparrow.FinishPredicativeFormula;
  end;
1752.
\langle \text{Parse expressions (parser.pas) } 1687 \rangle + \equiv
procedure CompleteAtomicFormula(var aParenthCnt : integer);
  var lPredSymbol: integer;
  label Predicate; { not actually used }
  begin (Parse left arguments in a formula 1754);
  case CurWord.Kind of
  sy\_Equal, PredicateSymbol: \langle Parse equation or (possibly infixed) predicate 1756 <math>\rangle;
  sy_Does, sy_Do: \(\rangle\) Parse formula with "does not" or "do not" 1757\);
  sy_Is: \langle \text{Parse formula with "is not" or "is not" } 1758 \rangle;
  othercases begin gSubexpPtr \uparrow. ProcessAtomicFormula; MissingWord(paWrongPredSymbol);
    gSubexpPtr \uparrow . InsertIncorrBasic;
    end;
  endcases;
  end:
1753.
        \langle \text{Error codes for parser 1675} \rangle + \equiv
  paWrongPredSymbol = 321;
1754. \langle \text{Parse left arguments in a formula } 1754 \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 1752.
```

1755. $\langle \text{Error codes for parser } 1675 \rangle + \equiv$ paFunctExp1 = 302;

544 FORMULAS Mizar Parser $\S1756$

```
\langle \text{Parse equation or (possibly infixed) predicate } 1756 \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 1752.
1757. \langle \text{Parse formula with "does not" or "do not" } 1757 \rangle \equiv
     begin gSubexpPtr\uparrow.ProcessDoesNot; ReadWord; Accept(sy_Not, paNotExpected);
     if CurWord.Kind \in [PredicateSymbol, sy\_Equal] then
         begin CompletePredicativeFormula(CurWord.Nr); gSubexpPtr<math>\uparrow.ProcessNegative;
         if CurWord.Kind \in [sy\_Equal, PredicateSymbol, sy\_Does, sy\_Do] then
               Complete Multi Predicative Formula
         end
     else begin gSubExpPtr\uparrow.InsertIncorrFormula; SynErr(CurPos, paInfinitiveExp)
         end;
    end
This code is used in section 1752.
                 \langle \text{ Parse formula with "is not" or "is not" } 1758 \rangle \equiv
     begin gSubexpPtr\uparrow.ProcessAtomicFormula; ReadWord;
     \textbf{if} \ (CurWord.Kind = sy\_Not) \land (AheadWord.Kind \in TermBegSys + [ModeSymbol, StructureSymbol, Symbol, StructureSymbol, Symbol, Symb
                   sy\_Set, AttributeSymbol, sy\_Non]) \lor (CurWord. Kind \in TermBeqSys + [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;
              end;
         othercases begin qSubexpPtr\uparrow.CompleteAttributes; qSubexpPtr\uparrow.FinishAttributiveFormula; end;
         endcases:
         end
     else begin qSubExpPtr\uparrow.InsertIncorrFormula; WrongWord(paTypeOrAttrExp);
         end;
     end
This code is used in section 1752.
1759. \langle \text{Error codes for parser } 1675 \rangle + \equiv
     paTypeOrAttrExp = 309;
```

 $\S1760$ Mizar Parser FORMULAS 545

```
1760.
               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) } 1687 \rangle + \equiv
procedure ViableFormula;
    var lParenthCnt: integer;
    label NotPrivate;
    begin qExpPtr\uparrow.CreateSubexpression; OpenParenth(lParenthCnt);
    case CurWord.Kind of
    sy\_For: UniversalFormula;
    sy\_Contradiction: begin gSubexpPtr\uparrow.ProcessContradiction; ReadWord; end;
    sy_Thesis: begin qSubexpPtr\f\.ProcessThesis; ReadWord; end;
    sy_Not: begin gSubexpPtr↑.ProcessNot; ReadWord; ViableFormula; KillSubexpression;
        gSubexpPtr \uparrow. ProcessNegative;
        end:
    Identifier: if AheadWord.Kind = sy\_LeftSquareBracket then \langle Parse private formula 1762 \rangle
        else goto NotPrivate;
    starts\_with\_term\_token:
        NotPrivate: begin gSubexpPtr \uparrow. StartAtomicFormula; { ???? TermSubexpression }
            GetClosedSubterm; CompleteArgument(lParenthCnt); CompleteAtomicFormula(lParenthCnt);
    InfixOperatorSymbol, PredicateSymbol, sy\_Does, 
        CompleteAtomicFormula(lParenthCnt);
    othercases begin qSubexpPtr\uparrow.InsertIncorrFormula; WrongWord(paWrongFormulaBeq)
    endcases; (Close parentheses for formula 1764);
    end;
              \langle \text{Error codes for parser } 1675 \rangle + \equiv
    paWrongFormulaBeq = 396;
1762. \langle \text{Parse private formula } 1762 \rangle \equiv
    begin gSubexpPtr\uparrow.StartPrivateFormula; ReadWord; ReadWord;
    if CurWord.Kind \neq sy\_RightSquareBracket then GetArguments(MaxVisArgNbr);
    Accept(sy\_RightSquareBracket, paRightSquareExp2); qSubexpPtr\uparrow.FinishPrivateFormula;
    end
This code is used in section 1760.
1763. \langle \text{Error codes for parser } 1675 \rangle + \equiv
    paRightSquareExp2 = 371;
1764. \langle Close parentheses for formula 1764 \rangle \equiv
    while lParenthCnt > 0 do
        begin Conditional Tail; gSubexpPtr\\\ .ProcessRightParenthesis;
        Accept(sy\_RightParanthesis, paRightParenthExp4); dec(lParenthCnt); CloseParenth(lParenthCnt);
        end
This code is used in section 1760.
1765. \langle Error codes for parser 1675 \rangle + \equiv
    paRightParenthExp4 = 370;
```

546 FORMULAS Mizar Parser $\S1766$

1766. Precedence for logical connectives. We will now "hardcode" the precedence for logical connectives 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) } \, 1687 \, \rangle \, + \equiv \\ \textbf{procedure } \, \textit{ConjunctiveTail}; \\ \textbf{begin while } \, (\textit{CurWord.Kind} = \textit{sy\_Ampersand}) \, \land \, (\textit{AheadWord.Kind} \neq \textit{sy\_Ellipsis}) \, \textbf{do} \\ \textbf{begin } \, g\textit{SubexpPtr} \uparrow . \textit{ProcessBinaryConnective}; \, \textit{ReadWord}; \, \textit{ViableFormula}; \, \textit{KillSubexpression}; \\ \, g\textit{SubexpPtr} \uparrow . \textit{FinishBinaryFormula}; \\ \textbf{end}; \\ \textbf{end}; \\ \end{cases}
```

1767. Mizar parses flexary conjunctions (" $\Phi[0]$ & ... & $\Phi[n]$ ") as weaker than "ordinary conjunction". For example " Ψ & $\Phi[0]$ & ... & $\Phi[n]$ " parses as "(Ψ & $\Phi[0]$) & ... & $\Phi[n]$ ".

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) } \, 1687 \rangle \, + \equiv \\ \textbf{procedure } \, \mathit{FlexConjunctiveTail}; \\ \textbf{begin } \, \mathit{ConjunctiveTail}; \\ \textbf{if } \, \mathit{CurWord.Kind} = \mathit{sy\_Ampersand then} \\ \textbf{begin } \, \mathit{Assert}(\mathit{AheadWord.Kind} = \mathit{sy\_Ellipsis}); \, \mathit{ReadWord}; \, \mathit{ReadWord}; \, \mathit{Accept}(\mathit{sy\_Ampersand}, 402); \\ \mathit{gSubexpPtr} \uparrow . \mathit{ProcessFlexConjunction}; \, \mathit{ViableFormula}; \, \mathit{ConjunctiveTail}; \, \mathit{KillSubexpression}; \\ \mathit{gSubexpPtr} \uparrow . \mathit{FinishFlexConjunction}; \\ \textbf{end}; \\ \textbf{end}; \\ \textbf{end};
```

1768. 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 \, {\it Parse expressions \,} ({\it parser.pas}) \, \, {\it 1687} \, \rangle + \equiv \ {\it procedure \,} \, {\it Disjunctive \,} {\it Tail;} \ {\it begin \,} \, {\it Flex \,} {\it Conjunctive \,} {\it Tail;} \ {\it while \,} \, ({\it CurWord.Kind = sy\_Or}) \wedge ({\it Ahead \,} {\it Word.Kind \neq sy\_Ellipsis}) \, {\it do} \ {\it begin \,} \, {\it gSubexp\,} {\it Ptr} \uparrow. {\it Process \,} {\it Binary \,} {\it Connective; \,} \, {\it Read \,} {\it Word; \,} \, {\it Viable \,} {\it Formula; \,} \, {\it Flex \,} {\it Conjunctive \,} {\it Tail; \,} \, {\it Kill \,} {\it Subexp\,} {\it Ptr} \uparrow. {\it Finish \,} {\it Binary \,} {\it Formula; \,} \, {\it end; \,} \, {\it end; \,} \,
```

 $\{1769 \quad \text{Mizar Parser}$ FORMULAS 547

1769. 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.

```
⟨ Parse expressions (parser.pas) 1687⟩ +≡
procedure FlexDisjunctiveTail;
begin DisjunctiveTail;
if CurWord.Kind = sy_Or then
begin Assert(AheadWord.Kind = sy_Ellipsis); ReadWord; ReadWord; Accept(sy_Or, 401);
gSubexpPtr↑.ProcessFlexDisjunction; ViableFormula; DisjunctiveTail; KillSubexpression;
gSubexpPtr↑.FinishFlexDisjunction;
end;
end;
```

1770. 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) 1687⟩ +≡
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;
1771. ⟨Error codes for parser 1675⟩ +≡
paUnexpConnective = 336;
```

1772. 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 Term \rangle \ \langle Qualifying\text{-}Segment \rangle : \langle Formula\text{-}Subexpression} \rangle \}
```

This will also serve as the "workhorse" for parsing a formula expression.

```
⟨ Parse expressions (parser.pas) 1687⟩ +≡
procedure FormulaSubexpression;
begin ViableFormula; ConditionalTail; KillSubexpression;
end;
```

Section 24.2. COMMUNICATION WITH ITEMS

1773. 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.

§1774 Mizar Parser MISCELLANEOUS 549

Section 24.3. MISCELLANEOUS

1774. Parsing a label. 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) 1774⟩ ≡
    { Miscellaneous }
procedure ProcessLab;
begin gItemPtr↑.ProcessLabel; {(§1545)}
if (CurWord.Kind = Identifier) ∧ (AheadWord.Kind = sy_Colon) then
begin ReadWord; ReadWord end;
end;
See also sections 1775, 1776, 1777, 1778, 1779, 1782, 1785, 1787, 1789, and 1790.
This code is used in section 1673.
```

1775. Telling the gItemPtr state variable we are about to parse a sentence just invokes the StartSentence (§1460) method, then the Parser parses the formula, and the gItemPtr "finishes" the sentence (which is an empty method).

```
⟨ Process miscellany (parser.pas) 1774⟩ +≡
procedure ProcessSentence;
begin gItemPtr↑.StartSentence; FormulaExpression; gItemPtr↑.FinishSentence;
end:
```

1776. 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}\ (\operatorname{parser.pas})\ 1774 \rangle +\equiv \\  \operatorname{procedure}\ InCorrSentence; \\  \operatorname{begin}\ gItemPtr \uparrow. StartSentence;\ gItemPtr \uparrow. CreateExpression(exFormula); \\  gExpPtr \uparrow. CreateSubexpression;\ gSubexpPtr \uparrow. InsertIncorrFormula;\ KillSubexpression;\ KillExpression; \\  gItemPtr \uparrow. FinishSentence; \\  \operatorname{end}; \\  \end{array}
```

1777. 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) 1774⟩ +≡
procedure InCorrStatement;
begin gItemPtr↑.ProcessLabel; gItemPtr↑.StartRegularStatement; InCorrSentence;
end:
```

550 MISCELLANEOUS Mizar Parser $\S1778$

1778. The Parser is looking at either

```
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 \{ \text{ and } \langle Hypotheses \rangle \}
That is to say, a bunch of (possibly labeled) formulas joined together by "and" keywords.
\langle Process miscellany (parser.pas) 1774 \rangle + \equiv
procedure ProcessHypotheses;
  begin repeat ProcessLab; ProcessSentence; qItemPtr↑.FinishHypothesis;
  until \neg Occurs(sy\_And)
  end;
1779.
          An assumption is either collective (using hypotheses) or singular (a single, possibly labeled, formula).
\langle Process miscellany (parser.pas) 1774 \rangle + \equiv
procedure Assumption;
  begin if CurWord.Kind = sy\_That then \langle Parse collective assumption 1780 \rangle
  else \langle \text{Parse singule assumption } 1781 \rangle;
  gItemPtr \uparrow . FinishAssumption;
  end;
1780. \langle \text{ Parse collective assumption } 1780 \rangle \equiv
  begin qItemPtr \uparrow .StartCollectiveAssumption; \{ (§1476) \} ReadWord; ProcessHypotheses
  end
This code is used in section 1779.
1781. \langle \text{ Parse singule assumption } 1781 \rangle \equiv
  begin ProcessLab; ProcessSentence; qItemPtr \uparrow. FinishHypothesis; { (§1540) }
  end
This code is used in section 1779.
          Fixed variables. 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 miscellany (parser.pas) 1774 \rangle + \equiv
procedure FixedVariables;
  begin gItemPtr \uparrow . StartFixedVariables;
  repeat (Parse segment of fixed variables 1783);
  until \neg Occurs(sy\_Comma);
  gItemPtr \uparrow . FinishFixedVariables;
  end;
```

§1783 Mizar Parser MISCELLANEOUS 551

1783. 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 } 1783 \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 1782.
1784.
       \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paIdentExp4 = 300;
1785. Parsing 'consider' statements. 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) 1774 \rangle + \equiv
procedure ProcessChoice;
  begin FixedVariables; Accept(sy\_Such, paSuchExp); Accept(sy\_That, paThatExp2);
  repeat gItemPtr \uparrow .StartCondition; ProcessLab; ProcessSentence; <math>gItemPtr \uparrow .FinishCondition;
  until \neg Occurs(sy\_And);
  gItemPtr\uparrow.FinishChoice;
  end:
1786. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paThatExp2 = 350; paSuchExp = 403;
1787. Parsing 'let' statements. 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) 1774 \rangle + \equiv
procedure Generalization;
  begin ReadWord; FixedVariables;
  if Occurs(sy_Such) then
     begin qItemPtr↑.StartAssumption; Accept(sy_That, paThatExp1); ProcessHypotheses;
     qItemPtr \uparrow . FinishAssumption;
```

end; end; 552 MISCELLANEOUS Mizar Parser §1788

```
1788. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paThatExp1 = 350;
1789. Parsing 'given' statements. 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 miscellary (parser.pas) 1774 \rangle + \equiv
procedure ExistentialAssumption;
  begin gBlockPtr \uparrow. CreateItem(itExistentialAssumption); ReadWord; ProcessChoice;
  end;
         The Parser is looking at either "canceled;" or "canceled \langle number \rangle;".
1790.
\langle Process miscellary (parser.pas) 1774 \rangle + \equiv
procedure Canceled;
  begin gBlockPtr\(\gamma\). CreateItem(itCanceled); ReadWord;
  if CurWord.Kind = Numeral then ReadWord;
  gItemPtr \uparrow. FinishTheorem;\\
  end;
```

§1791 Mizar Parser SIMPLE JUSTIFICATIONS 553

Section 24.4. SIMPLE JUSTIFICATIONS

1791. 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) } 1791 \rangle \equiv
    { Simple Justifications }
procedure GetReferences;
  begin gItemPtr\uparrow.StartReferences;
  repeat ReadWord; \langle Parse single reference 1792 \rangle;
  until CurWord.Kind \neq sy\_Comma;
  gItemPtr\uparrow.FinishReferences;
  end:
See also sections 1796 and 1800.
This code is used in section 1673.
1792. \langle \text{ Parse single reference } 1792 \rangle \equiv
  case CurWord.Kind of
  MMLIdentifier: (Parse library references 1794);
  Identifier: begin gItemPtr↑.ProcessPrivateReference; ReadWord end;
  othercases WrongWord(paWrongReferenceBeg);
  endcases
This code is used in section 1791.
1793. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paWrongReferenceBeg = 308;
1794. 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 "\langle Article \rangle" instead of
"\(\lambda rticle \rangle : \lambda number \rangle\" \rightarron " \lambda 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 } 1794 \rangle \equiv
  begin gItemPtr↑.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
This code is used in section 1792.
1795. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paNumExp = 307; paDefExp = 312; paColonExp4 = 384;
```

554 SIMPLE JUSTIFICATIONS Mizar Parser §1796

1796. 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)} 1791 \rangle + \equiv
procedure GetSchemeReference;
  begin gItemPtr \uparrow .StartSchemeReference; ReadWord;
  case CurWord.Kind of
  MMLIdentifier: (Parse reference to scheme from MML 1798);
  Identifier: begin gItemPtr\uparrow.ProcessSchemeReference; ReadWord end;
  othercases WrongWord(paWrongReferenceBeg);
  endcases;
  if CurWord.Kind = sy\_LeftParanthesis then
    \textbf{begin} \ \ GetReferences; \ \ Accept(sy\_RightParanthesis,paRightParenthExp7)
  gItemPtr\uparrow.FinishSchemeReference;
  end;
1797.
         \langle \text{Error codes for parser 1675} \rangle + \equiv
  paRightParenthExp7 = 370;
1798. 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 1798\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);
       Read Word;
       end
    else ErrImm(paSchExp);
    qItemPtr\uparrow.ProcessSchemeNumber; Accept(Numeral, paNumExp);
  else MissingWord(paColonExp4);
  gItemPtr \uparrow . FinishSchLibraryReferences;
  end
This code is used in section 1796.
       \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paSchExp = 313;
```

 $\S1800$ Mizar Parser SIMPLE JUSTIFICATIONS 555

1800. 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.

⟨Parse simple justifications (parser.pas) 1791⟩ +≡
procedure SimpleJustification;
begin gItemPtr↑.StartSimpleJustification;
case CurWord.Kind of
sy_By: GetReferences;
sy_Semicolon, sy_DotEquals:;
sy_From: GetSchemeReference;
othercases WrongWord(paWrongJustificationBeg);
endcases; gItemPtr↑.FinishSimpleJustification;
end;
1801. ⟨Error codes for parser 1675⟩ +≡

paWrongJustificationBeg = 395;

Section 24.5. STATEMENTS AND REASONINGS

1802. 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)} 1802 \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 1803, 1804, 1805, 1815, 1816, 1821, 1830, and 1832.
This code is used in section 1673.
1803. 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) 1802⟩ +≡
procedure Justification;
  begin qItemPtr \uparrow .StartJustification;
  case CurWord.Kind of
  sy_Proof: parse_proof;
  othercases SimpleJustification;
  endcases; gItemPtr\uparrow.FinishJustification;
  end:
```

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 $\neg Occurs(sy_Comma)$ $\langle \text{Parse statements and reasoning (parser.pas)} 1802 \rangle + \equiv$ **procedure** ReadTypeList; begin case CurWord.Kind of $sy_RightSquareBracket$, $sy_RightParanthesis$:; **othercases** parse_comma_separated_types; endcases; end: **1805.** 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 $\langle \text{Parse statements and reasoning (parser.pas)} 1802 \rangle + \equiv$ **procedure** RegularStatement; forward; $\{(\S1832)\}$ procedure PrivateItem; **begin** $qBlockPtr\uparrow.ProcessLink$; if $CurWord.Kind = sy_Then$ then ReadWord; case CurWord.Kind of $sy_Deffunc: \langle Parse a "deffunc" 1807 \rangle;$ sy_Defpred: (Parse a "defpred" 1809); sy_Set : $\langle Parse a "set" constant definition 1811 <math>\rangle$; *sy_Reconsider*: ⟨ Parse a "reconsider" statement 1813⟩; $sy_Consider$: **begin** $gBlockPtr\uparrow.CreateItem(itChoice)$; ReadWord; ProcessChoice; SimpleJustification; end: other_regular_statements: **begin** qBlockPtr\.CreateItem(itRegularStatement); RegularStatement; **end**; othercases begin $gBlockPtr \uparrow$. CreateItem(itIncorrItem); WrongWord(paWrongItemBeg); end; endcases; end: **1806.** $\langle \text{Error codes for parser } 1675 \rangle + \equiv$ paWrongItemBeg = 391;1807. $\langle Parse \ a "deffunc" \ 1807 \rangle \equiv$ **begin** $qBlockPtr\uparrow.CreateItem(itPrivFuncDefinition); ReadWord; <math>qItemPtr\uparrow.StartPrivateDefiniendum;$ $Accept(Identifier, paIdentExp6); Accept(sy_LeftParanthesis, paLeftParenthExp); ReadTypeList;$ $Accept(sy_RightParanthesis, paRightParenthExp8); gItemPtr\uparrow.StartPrivateDefiniens;$ $Accept(sy_Equal, paEqualityExp1); TermExpression; gItemPtr \uparrow. FinishPrivateFuncDefinienition;$ end This code is used in section 1805. **1808.** $\langle \text{Error codes for parser } 1675 \rangle + \equiv$

paIdentExp6 = 300; paLeftParenthExp = 360; paRightParenthExp8 = 370; paEqualityExp1 = 380;

```
1809.
        \langle \text{Parse a "defpred" 1809} \rangle \equiv
  begin gBlockPtr \uparrow. CreateItem(itPrivPredDefinition); ReadWord; <math>gItemPtr \uparrow. StartPrivateDefiniendum;
  Accept(Identifier, paIdentExp7); Accept(sy\_LeftSquareBracket, paLeftSquareExp); ReadTypeList;
  Accept(sy\_RightSquareBracket, paRightSquareExp4); gItemPtr \uparrow. StartPrivateDefiniens;
  Accept(sy\_Means, paMeansExp); FormulaExpression; gItemPtr \uparrow . FinishPrivatePredDefinienition;
  end
This code is used in section 1805.
1810. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paIdentExp7 = 300; paLeftSquareExp = 361; paRightSquareExp4 = 371; paMeansExp = 386;
1811. \langle \text{Parse a "set" constant definition 1811} \rangle \equiv
  begin gBlockPtr \uparrow. CreateItem(itConstantDefinition); ReadWord;
  repeat gItemPtr\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 1805.
1812. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paIdentExp8 = 300; paEqualityExp2 = 380;
1813. \langle \text{Parse a "reconsider" statement 1813} \rangle \equiv
  begin gBlockPtr\uparrow.CreateItem(itReconsider); ReadWord;
  repeat gItemPtr\uparrow.ProcessReconsideredVariable; Accept(Identifier, paIdentExp9);
     case CurWord.Kind of
     sy\_Equal: begin ReadWord; TermExpression; gItemPtr\uparrow.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 1805.
1814. \langle Error codes for parser 1675 \rangle + \equiv
  paIdentExp9 = 300; paAsExp = 388;
1815. 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)} 1802 \rangle + \equiv
procedure ProcessPragmas;
  begin while CurWord.Kind = Pragma do
     begin SetParserPragma(CurWord.Spelling); \{(\S1375)\}
     gBlockPtr\uparrow.ProcessPragma; \{(\S1395)\}
     ReadTokenProc;
     end:
  end;
```

This code is used in section 1817.

```
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".
⟨ Parse statements and reasoning (parser.pas) 1802⟩ +≡
procedure LinearReasoning;
  begin while CurWord.Kind \neq sy\_End do
     begin StillCorrect \leftarrow true; ProcessPragmas; \langle Parse statement of linear reasoning 1817 \rangle;
     Semicolon;
     end;
  end;
         Most statements are delegated to their own dedicated function.
1817.
\langle Parse statement of linear reasoning 1817 \rangle \equiv
  case CurWord.Kind of
  sy\_Let: \mathbf{begin} \ gBlockPtr \uparrow. CreateItem(itGeneralization); \ Generalization; \mathbf{end};
  sy_Given: ExistentialAssumption;
  sy\_Assume: begin gBlockPtr\uparrow. CreateItem(itAssumption); ReadWord; Assumption; end;
  sy_Take: \langle Parse "take" statement for linear reasoning 1818 \rangle;
  sy\_Hereby: begin qBlockPtr\uparrow. CreateItem(itConclusion); Reasoning; end;
  (Parse "thus" and "hence" for linear reasoning 1819);
  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 1820 \rangle;
  othercases PrivateItem;
  endcases
This code is used in section 1816.
         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 1818} \rangle \equiv
  begin gBlockPtr↑. CreateItem(itExemplification); ReadWord;
  repeat if (CurWord.Kind = Identifier) \land (AheadWord.Kind = sy\_Equal) then
       begin qItemPtr↑.ProcessExemplifyinqVariable; ReadWord; ReadWord; TermExpression;
       qItemPtr\uparrow.FinishExemplifyingVariable;
       end
     else begin qItemPtr\uparrow.StartExemplifyingTerm; TermExpression; qItemPtr\uparrow.FinishExemplifyingTerm;
  until \neg Occurs(sy\_Comma);
  end
```

if $CurWord.Kind = sy_Per$ then NonBlockReasoning;

KillBlock; AcceptEnd(CasePos); Semicolon;

This code is used in section 1821.

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 qBlockPtr state variable "primes the pump" by creating a "conclusion" statement. $\langle \text{Parse "thus" and "hence" for linear reasoning 1819} \rangle \equiv$ sy_Hence : **begin** $qBlockPtr\uparrow.ProcessLink$; ReadWord; $qBlockPtr\uparrow.CreateItem(itConclusion)$; Regular Statement;end: sy_Thus: **begin** ReadWord; gBlockPtr\\capp.ProcessLink; if $CurWord.Kind = sy_Then$ then ReadWord; $gBlockPtr\uparrow.CreateItem(itConclusion); RegularStatement;$ end This code is used in section 1817. 1820. Parsing 'then' linked statements. $\langle \text{ Parse "then" for linear reasoning 1820} \rangle \equiv$ begin if $AheadWord.Kind = sy_Per$ then begin $gBlockPtr\uparrow.ProcessLink; ReadWord; exit;$ end **else** PrivateItem; end This code is used in section 1817. 1821. Non-block Reasoning. The Parser has just encountered a "per cases" statement. Now it must parse "suppose" items. $\langle \text{Parse statements and reasoning (parser.pas)} 1802 \rangle + \equiv$ **procedure** NonBlockReasoning; var CasePos: Position; lCaseKind: TokenKind; \(\rangle \text{Process "case"} \) (local procedure) 1822 \(\rangle ; \) 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 1823); if $(CurWord.Kind \neq sy_Case) \land (CurWord.Kind \neq sy_Suppose)$ then Try to synchronize after failing to find initial 'case' or 'suppose' 1825); repeat (Parse "suppose" or "case" block 1827); **until** $(Curword.Kind = sy_End);$ end: endcases: end: 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). $\langle Process "case" (local procedure) 1822 \rangle \equiv$ **procedure** *ProcessCase*; **begin** Assumption; Semicolon; LinearReasoning;

1823. 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.

```
1824. \langle \text{Error codes for parser } 1675 \rangle + \equiv paPerExp = 231; paCasesExp = 351;
```

1826. $\langle \text{Error codes for parser } 1675 \rangle + \equiv$

1825. 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.

```
⟨ Try to synchronize after failing to find initial 'case' or 'suppose' 1825⟩ ≡ begin MissingWord(paSupposeOrCaseExp); lCaseKind \leftarrow sy\_Suppose; gBlockPtr\uparrow.CreateItem(itCaseBlock); gBlockPtr\uparrow.CreateIlock(blSuppose); gBlockPtr\uparrow.CreateItem(itSupposeHead); StillCorrect \leftarrow true; CasePos \leftarrow CurPos; ProcessCase; end

This code is used in section 1821.
```

```
paSupposeOrCaseExp = 232;

1827.    ⟨Parse "suppose" or "case" block 1827⟩ ≡
    while (CurWord.Kind = sy_Case) ∨ (CurWord.Kind = sy_Suppose) do
    ⟨Parse contents of "suppose" block 1828⟩;
    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 1829⟩;
    endcases
```

This code is used in section 1821.

1831. $\langle \text{Error codes for parser } 1675 \rangle + \equiv$

paProofExp = 389;

1828. 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. **define** $create_supposition_block \equiv$ 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 $gBlockPtr\uparrow.CreateItem(itSupposeHead)$ \langle Parse contents of "suppose" block 1828 $\rangle \equiv$ **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 1827. $\langle \text{Synchronize after missing 'suppose' or 'case' token 1829} \rangle \equiv$ **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 1827. 1830. Reasoning. The Parser is looking at "proof", "hereby", or "now". The syntax for Mizar says 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). $\langle \text{ Parse statements and reasoning (parser.pas) } 1802 \rangle + \equiv$ **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** $gBlockPtr\uparrow$. 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;

```
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.
\langle \text{Parse statements and reasoning (parser.pas)} 1802 \rangle + \equiv
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 1833);
     othercases begin gItemPtr \uparrow . StartJustification; SimpleJustification; <math>gItemPtr \uparrow . FinishJustification;
       gItemPtr \uparrow . FinishCompactStatement;
       while CurWord.Kind = sy\_DotEquals do \langle Parse iterative equations 1834 \rangle;
       end;
     endcases;
     end;
  endcases;
  end;
        \langle \text{Parse "proof" block 1833} \rangle \equiv
1833.
  begin qItemPtr\uparrow.StartJustification;
  if ProofPragma then Reasoning
  else IgnoreProof;
  gItemPtr \uparrow . FinishJustification;
This code is used in section 1832.
1834. \langle Parse iterative equations 1834 \rangle \equiv
  begin qItemPtr\uparrow.StartIterativeStep; ReadWord; TermExpression; qItemPtr\uparrow.ProcessIterativeStep;
  qItemPtr \uparrow . StartJustification; SimpleJustification; qItemPtr \uparrow . FinishJustification;
  gItemPtr\uparrow.FinishIterativeStep;
This code is used in section 1832.
```

564 PATTERNS Mizar Parser $\S1835$

Section 24.6. PATTERNS

1835. 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) } 1835 \rangle \equiv
     { Patterns }
\mathbf{var}\ g\mathit{VisibleNbr}\colon \mathit{integer};
procedure GetVisible;
  begin gItemPtr \uparrow .ProcessVisible; \{ (\S1505) \}
  inc(gVisibleNbr); Accept(Identifier, paIdentExp3);
  end;
See also sections 1837, 1838, 1843, 1845, 1847, 1853, 1856, and 1858.
This code is used in section 1673.
1836.
          \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paIdentExp3 = 300;
1837. We will need to Parse a comma-separated list of identifiers when determining a pattern.
\langle \text{Parse patterns (parser.pas)} 1835 \rangle + \equiv
procedure ReadVisible;
  begin gItemPtr\uparrow.StartVisible; gVisibleNbr \leftarrow 0;
  repeat GetVisible;
  until \neg Occurs(sy\_Comma);
  gItemPtr\uparrow.FinishVisible;
  end;
1838. 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.
\langle \text{Parse patterns (parser.pas) } 1835 \rangle + \equiv
procedure GetModePattern;
```

```
⟨ Parse patterns (parser.pas) 1835⟩ +≡
procedure GetModePattern;
var lModesymbol: integer;
begin gItemPtr↑.StartModePattern; { (§1493) }
case CurWord.Kind of
sy_Set: ⟨ Parse pattern for "set" as a mode 1840⟩;
ModeSymbol: ⟨ Parse pattern for a mode symbols 1842⟩
othercases WrongWord(paWrongModePatternBeg);
endcases;
gItemPtr↑.FinishModePattern; { (§1494) }
end;

1839. ⟨ Error codes for parser 1675⟩ +≡
paWrongModePatternBeg = 303;
```

 $\{1840$ Mizar Parser PATTERNS 565

```
\langle \text{ Parse pattern for "set" as a mode 1840} \rangle \equiv
1840.
  begin if AheadWord.Kind = sy\_Of then WrongWord(paWrongModePatternSet)
  else ReadWord;
  end
This code is used in section 1838.
1841. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paWrongModePatternSet = 315;
1842. 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 qVisibleNbr state variable's current
value.
  define qet\_index\_compare\_to\_default(\#) \equiv [\#] = \$FF
  define entry\_is\_unitialized(\#) \equiv \#.fList \uparrow get\_index\_compare\_to\_default
\langle \text{ Parse pattern for a mode symbols } 1842 \rangle \equiv
  begin lModeSymbol \leftarrow CurWord.Nr; gVisibleNbr \leftarrow 0; ReadWord; gItemPtr<math>\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 qVisibleNbr;
  end
This code is used in section 1838.
1843. Parsing the visible arguments for a functor relies on this helper function.
\langle \text{Parse patterns (parser.pas)} 1835 \rangle + \equiv
procedure ReadParams;
  begin if Occurs(sy_LeftParanthesis) then
     begin ReadVisible; Accept(sy_RightParanthesis, paRightParenthExp5)
  else if CurWord.Kind = Identifier then
       begin gItemPtr\uparrow.StartVisible; GetVisible; gItemPtr\uparrow.FinishVisible; end;
  end;
1844. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paRightParenthExp5 = 370;
```

566 PATTERNS Mizar Parser $\S1845$

1845. Attribute patterns allows for arguments *only on the right* of the attribute symbol, i.e., something like

$$\underbrace{\langle Identifier \rangle \text{ is } \langle Arguments \rangle \ \langle Attribute\text{-}Name \rangle}_{\text{pattern}} \text{ means} \dots$$

```
\langle \text{Parse patterns (parser.pas)} 1835 \rangle + \equiv
procedure GetAttrPattern;
  begin gItemPtr\uparrow.StartAttributePattern; <math>gVisibleNbr \leftarrow 0; GetVisible;
  gItemPtr\uparrow.ProcessAttributePattern; Accept(sy\_Is, paIsExp);
  if Occurs(sy\_LeftParanthesis) then
     begin ReadVisible; Accept(sy_RightParanthesis, paRightParenthExp11)
     end
  else if CurWord.Kind = Identifier then ReadVisible;
  gItemPtr\uparrow.FinishAttributePattern; Accept(AttributeSymbol, paAttrExp2);
  end;
1846.
          \langle \text{Error codes for parser 1675} \rangle + \equiv
  paAttrExp2 = 306; paRightParenthExp11 = 370; paIsExp = 383;
1847.
         Functor patterns generically look like:
                                  func \langle Arguments \rangle \langle Identifier \rangle \langle Arguments \rangle -> . . .
or
                              \verb|func| \langle \textit{Left-Bracket} \rangle \ \langle \textit{Arguments} \rangle \ \langle \textit{Right-Bracket} \rangle \ -> \dots
                                                            pattern
\langle \text{Parse patterns (parser.pas) } 1835 \rangle + \equiv
procedure GetFuncPattern;
  begin qItemPtr \uparrow .StartFunctorPattern;
  case CurWord.Kind of
  Identifier, InfixOperatorSymbol, sy\_LeftParanthesis: \langle \ Parse \ infix \ functor \ pattern \ 1849 \ \rangle;
  LeftCircumfixSymbol, sy\_LeftSquareBracket, sy\_LeftCurlyBracket: \langle Parse bracket functor pattern 1851 \rangle;
  othercases begin WrongWord(paWrongFunctorPatternBeg); qItemPtr\uparrow.FinishFunctorPattern; end;
  endcases:
  end;
          \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paWrongFunctorPatternBeg = 399;
1849. \langle \text{ Parse infix functor pattern } 1849 \rangle \equiv
  begin ReadParams; gItemPtr\uparrow.ProcessFunctorSymbol; { (§1502) }
  Accept(InfixOperatorSymbol, paFunctExp2); ReadParams; gItemPtr\uparrow.FinishFunctorPattern;
  end
```

This code is used in section 1847.

1850. $\langle \text{Error codes for parser } 1675 \rangle + \equiv paFunctExp2 = 302;$

 $\S1851$ Mizar Parser PATTERNS 567

```
1851.
         \langle \text{ Parse bracket functor pattern } 1851 \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 1847.
1852. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paRightBraExp2 = 310;
        Predicate patterns resemble infix functor patterns.
\langle \text{Parse patterns (parser.pas) } 1835 \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 1855 \rangle;
  othercases WrongWord(paWrongPredPattern);
  endcases; gItemPtr\uparrow.FinishPredicatePattern;
  end;
1854. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paWrongPredPattern = 301;
1855. \langle \text{Parse predicate pattern } 1855 \rangle \equiv
  begin lPredSymbol \leftarrow CurWord.Nr;
  if CurWord.Kind = sy\_Equal then lPredSymbol \leftarrow EqualitySym;
  gVisibleNbr \leftarrow 0; ReadWord;
  if CurWord.Kind = Identifier then ReadVisible;
  if (PredMaxArgs.fList\uparrow[lPredSymbol] < qVisibleNbr)\lor(entry\_is\_uninitialized(PredMaxArgs)(lPredSymbol))
          then PredMaxArgs.fList \uparrow [lPredSymbol] \leftarrow gVisibleNbr;
  end
This code is used in section 1853.
         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) } 1835 \rangle + \equiv
procedure Specification;
  begin gItemPtr\uparrow.StartSpecification; Accept(sy\_Arrow, paArrowExp1); TypeExpression;
  gItemPtr \uparrow . FinishSpecification;
  end;
1857. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paArrowExp1 = 385;
```

568 PATTERNS Mizar Parser $\S1858$

1858. 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) } 1835 \rangle + \equiv
procedure GetStructPatterns;
  var lStructureSymbol: integer;
  begin gBlockPtr↑. CreateItem(itDefStruct); ReadWord;
  Parse ancestors of structure, if there are any 1859;
  ⟨ Parse "over" and any structure arguments, if any 1861⟩;
  gItemPtr \uparrow . StartFields;
  (Update max arguments for structure symbol, if needed 1863);
  ⟨ Parse the fields of the structure definition 1864⟩;
  end;
1859.
         \langle \text{Parse ancestors of structure, if there are any 1859} \rangle \equiv
  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 1858.
1860. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paRightParenthExp6 = 370;
1861. (Parse "over" and any structure arguments, if any 1861) \equiv
  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 1858.
1862. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paStructExp1 = 304;
1863. (Update max arguments for structure symbol, if needed 1863) \equiv
  if lStructureSymbol \neq \$FF then
     if (StructModeMaxArgs.fList\uparrow[lStructureSymbol] < gVisibleNbr) \lor
            (entry\_is\_uninitialized(StructModeMaxArgs)(lStructureSymbol)) then
       StructModeMaxArgs.fList\uparrow[lStructureSymbol] \leftarrow gVisibleNbr
This code is used in section 1858.
1864. (Parse the fields of the structure definition 1864) \equiv
  Accept(sy_StructLeftBracket, paLeftDoubleExp3);
  repeat (Parse field for the structure definition 1866);
  until \neg Occurs(sy\_Comma);
  gItemPtr \uparrow . FinishFields; Accept(sy\_StructRightBracket, paRightDoubleExp2)
This code is used in section 1858.
```

 $\S1865$ Mizar Parser PATTERNS 569

```
1865. ⟨Error codes for parser 1675⟩ +≡ paLeftDoubleExp3 = 363; paRightDoubleExp2 = 373;
1866. ⟨Parse field for the structure definition 1866⟩ ≡ gItemPtr↑.StartAggrPattSegment; repeat gItemPtr↑.ProcessField; Accept(SelectorSymbol, paSelectExp1); until ¬Occurs(sy_Comma); Specification; gItemPtr↑.FinishAggrPattSegment
This code is used in section 1864.
1867. ⟨Error codes for parser 1675⟩ +≡ paSelectExp1 = 305;
```

570 DEFINITIONS Mizar Parser $\S1868$

Section 24.7. DEFINITIONS

1868. 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) } 1868 \rangle \equiv
     \{ Definitions \}
procedure Construction Type;
  begin gItemPtr\uparrow.StartConstructionType; \{ (\S1544) \}
  if CurWord.Kind = sy\_Arrow then
     begin ReadWord; TypeExpression end;
  gItemPtr\uparrow.FinishConstructionType; \{(\S1486)\}
  end:
See also sections 1869, 1870, 1880, 1881, 1883, 1885, 1887, 1892, 1895, 1898, 1904, 1906, 1908, 1910, 1911, 1913, and 1914.
This code is used in section 1673.
1869. Parsing correctness conditions amounts to looping through every "\( \langle Correctness \rangle \langle Justification \rangle;"
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) } 1868 \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; { (§1543) 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;
```

§1870 Mizar Parser DEFINITIONS 571

```
1870.
\langle \text{Parse definitions (parser.pas) } 1868 \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 1871 \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 \(\frac{1872}{2}\rangle\);
  Semicolon; Correctness;
  while (CurWord.Kind = sy\_Property) do
     begin gBlockPtr \uparrow. CreateItem(itProperty); StillCorrect \leftarrow true; ReadWord; Justification; Semicolon;
     end;
  gBlockPtr \uparrow . FinishDefinition;
  end;
1871. \langle Parse mode definition 1871 \rangle \equiv
  begin gBlockPtr\uparrow.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 1870.
1872. \langle \text{Parse definiens } 1872 \rangle \equiv
  case CurWord.Kind of
  sy\_Means: \langle Parse "means" definiens 1873 \rangle;
  sy\_Equals: \langle Parse "equals" definiens 1877 <math>\rangle;
  endcases
This code is used in section 1870.
```

572 DEFINITIONS Mizar Parser §1873

```
1873.
        \langle \text{Parse "means" definiens 1873} \rangle \equiv
  begin gItemPtr\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 1875 \rangle
  else gItemPtr\uparrow.FinishOtherwise;
  gItemPtr \uparrow . FinishDefiniens;
  end
This code is used in section 1872.
1874. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paIdentExp10 = 300; paColonExp2 = 384;
1875. \langle \text{Parse "means" definition-by-cases 1875} \rangle \equiv
  begin qItemPtr \uparrow .StartGuard; ReadWord; FormulaExpression; <math>qItemPtr \uparrow .FinishGuard;
  while Occurs(sy\_Comma) do
     begin FormulaExpression; gItemPtr\uparrow.StartGuard; Accept(sy\_If, paIfExp); FormulaExpression;
     gItemPtr\uparrow.FinishGuard;
     end;
  if CurWord.Kind = sy\_Otherwise then
     begin gItemPtr\uparrow.StartOtherwise; ReadWord; FormulaExpression; gItemPtr\uparrow.FinishOtherwise; end;
  end
This code is used in section 1873.
1876. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paIfExp = 381;
1877. \langle \text{Parse "equals" definiens 1877} \rangle \equiv
  if lDefKind \neq sy-Func then
     begin WrongWord(paUnexpEquals); end
  else begin gItemPtr↑.ProcessEquals; ReadWord;
     if Occurs(sy\_Colon) then
       begin gItemPtr↑.ProcessDefiniensLabel; Accept(Identifier, paIdentExp10);
       Accept(sy\_Colon, paColonExp2);
       end
     else gItemPtr\uparrow.ProcessDefiniensLabel;
     gItemPtr \uparrow . StartEquals; TermExpression;
     if CurWord.Kind = sy\_If then \langle Parse "equals" definition-by-cases 1879 \rangle
     else gItemPtr\uparrow.FinishOtherwise;
     gItemPtr\uparrow.FinishDefiniens;
     end
This code is used in section 1872.
1878. \langle Error codes for parser 1675 \rangle + \equiv
  paUnexpEquals = 186;
```

 $\{1879 \quad \text{Mizar Parser} \quad \text{DEFINITIONS} \quad 573$

```
1879.
         \langle \text{Parse "equals" definition-by-cases 1879} \rangle \equiv
  begin qItemPtr \uparrow .StartGuard; ReadWord; FormulaExpression; qItemPtr \uparrow .FinishGuard;
  while Occurs(sy\_Comma) do
    begin TermExpression; gItemPtr \uparrow . StartGuard; Accept(sy\_If, paIfExp); FormulaExpression;
    qItemPtr\uparrow.FinishGuard;
    end;
  if CurWord.Kind = sy\_Otherwise then
    begin qItemPtr\uparrow.StartOtherwise; ReadWord; TermExpression; qItemPtr\uparrow.FinishOtherwise;
    end:
  end
This code is used in section 1877.
1880. 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 = Symbol))
              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) } 1868 \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:
```

574 DEFINITIONS Mizar Parser §1881

```
The Parser is looking at the "synonym" token when this procedure is invoked.
\langle \text{Parse definitions (parser.pas) } 1868 \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 }
    qBlockPtr\uparrow.CreateItem(itAttrSynonym);\ GetAttrPattern;\ qItemPtr\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 }
    gBlockPtr \uparrow. CreateItem(itPredSynonym); gItemPtr \uparrow. StartDefPredicate; GetPredPattern;
    gItemPtr\uparrow.ProcessPredSynonym; Accept(sy\_For, paForExp); GetPredPattern;
    end
  othercases begin qBlockPtr\uparrow. CreateItem(itIncorrItem); ErrImm(paWronqPattBeq1);
  endcases;
  end;
        \langle Error codes for parser 1675\rangle + \equiv
  paWrongPattBeg1 = 314; paForExp = 382;
       Antonyms only make sense for attributes and predicates. A 314 error is raised for any other kind
1883.
of antonym.
\langle \text{Parse definitions (parser.pas) } 1868 \rangle + \equiv
procedure Antonym;
  begin ReadWord;
  case CurrPatternKind of
  Attributesymbol: begin
                             { Attribute antonym }
    gBlockPtr\uparrow.CreateItem(itAttrAntonym);\ GetAttrPattern;\ gItemPtr\uparrow.ProcessAttrAntonym;
    Accept(sy\_For, paForExp); GetAttrPattern;
    end;
  PredicateSymbol: begin
                              { Predicate antonym }
    qBlockPtr\uparrow.CreateItem(itPredAntonym);\ qItemPtr\uparrow.StartDefPredicate;\ GetPredPattern;
    qItemPtr\uparrow.ProcessPredAntonym; Accept(sy\_For, paForExp); GetPredPattern;
  othercases begin qBlockPtr\uparrow.CreateItem(itIncorrItem); ErrImm(paWronqPattBeq2);
    end;
  endcases;
  end:
1884. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paWrongPattBeg2 = 314;
```

 $\{1885 \quad \text{Mizar Parser} \quad \text{DEFINITIONS} \quad 575$

```
1885.
\langle \text{Parse definitions (parser.pas) } 1868 \rangle + \equiv
procedure UnexpectedItem;
  begin case CurWord.Kind of
  sy\_Case, sy\_Suppose, sy\_Hereby: begin ErrImm(paWrongItemBeg); 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;
  othercases begin ErrImm(paUnexpItemBeg); StillCorrect \leftarrow true; PrivateItem; end;
  endcases;
  end;
1886. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paUnexpItemBeg = 392;
         The Parser is currently looking at the "definition" token, so it will construct a definition block
1887.
AST.
\langle \text{Parse definitions (parser.pas) } 1868 \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 1888 \rangle;
  KillBlock; AcceptEnd(DefPos);
  end;
        \langle \text{ Parse item in definition block 1888} \rangle \equiv
  begin StillCorrect \leftarrow true; gBlockPtr \uparrow. ProcessRedefine;
  if Occurs(sy_Redefine) then (Check we are redefining a mode, attribute, functor, or predicate 1889);
  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 1891);
     Semicolon;
     end;
  endcases;
  end
This code is used in section 1887.
1889. Check we are redefining a mode, attribute, functor, or predicate 1889 \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 1888.
1890. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paUnexpRedef = 273;
```

576 DEFINITIONS Mizar Parser $\S1891$

```
1891.
         \langle \text{Parse loci, assumptions, unexpected items in a definition block 1891} \rangle \equiv
  case CurWord.Kind of
  sy\_Let: \mathbf{begin} \ qBlockPtr\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 1888.
        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) } 1868 \rangle + \equiv
procedure NotationBlock:
  var DefPos: Position;
  begin qBlockPtr\uparrow.CreateItem(itDefinition); <math>qBlockPtr\uparrow.CreateBlock(blNotation); DefPos \leftarrow CurPos;
  ReadWord;
  while CurWord.Kind \neq sy\_End do \langle Parse item for notation block 1893 \rangle;
  KillBlock; AcceptEnd(DefPos);
  end;
1893. \langle \text{Parse item for notation block 1893} \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 1894);
  endcases;
  end
This code is used in section 1892.
1894. \langle Parse semicolon-separated items in a notation block 1894\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 1893.
```

 $\S1895$ Mizar Parser DEFINITIONS 577

1895.

```
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) } 1868 \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 1897⟩;
    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 qSubexpPtr↑.ProcessAttribute; SynErr(CurPos, paAttrExp3);
         end;
       break;
       end;
    end;
  gSubexpPtr \uparrow. CompleteAttributes;
  end:
1896.
        \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paAttrExp3 = 306;
1897. (Parse arguments for attribute expression 1897) \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;
    gSubexpPtr\uparrow.StartAttributeArguments;\ ProcessArguments;\ gSubexpPtr\uparrow.FinishAttributeArguments;
    end
```

This code is used in section 1895.

578 DEFINITIONS Mizar Parser §1898

```
1898.
        Registration clusters.
\langle \text{Parse definitions (parser.pas) } 1868 \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; \{ (\S1458) \}
  gItemPtr\uparrow.CreateExpression(exAdjectiveCluster); \{(\S1531)\}
  gExpPtr\uparrow.CreateSubexpression; ATTSubexpression(lExpKind);
  case lExpKind of
  exTerm: gSubexpPtr \uparrow. CompleteClusterTerm;
  exNull, exAdjectiveCluster: gSubexpPtr \uparrow. CompleteAdjectiveCluster;
  endcases;
  KillSubexpression; KillExpression;
  case lExpKind of
  exTerm: (Parse functor registration cluster 1900);
  exNull, exAdjectiveCluster: case CurWord.Kind of
     sy\_Arrow: \langle Parse conditional registration cluster 1902 \rangle;
     sy\_For: \langle Parse existential registration cluster 1903 <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:
1899.
         \langle \text{Error codes for parser 1675} \rangle + \equiv
  paForOrArrowExpected = 406;
1900. \langle Parse functor registration cluster | 1900\rangle \equiv
  begin qItemPtr\(\frac{1}{2}\). FinishClusterTerm; Accept(sy_Arrow, paArrowExp2);
  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;
  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;
  gItemPtr \uparrow . FinishClusterType;
  end
This code is used in section 1898.
1901. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paAdjClusterExp = 223; paArrowExp2 = 385;
```

 $\S1902$ Mizar Parser DEFINITIONS 579

```
1902.
         \langle \text{ Parse conditional registration cluster } 1902 \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);
  gExpPtr\uparrow.CreateSubexpression;\ gSubexpPtr\uparrow.StartType;\ gSubexpPtr\uparrow.StartAttributes;
  GetAdjectiveCluster; RadixTypeSubexpression; gSubexpPtr \uparrow. CompleteAttributes;
  gSubexpPtr\uparrow.CompleteType;\ gSubexpPtr\uparrow.CompleteClusterType;\ KillSubexpression;\ KillExpression;
  gItemPtr\uparrow.FinishClusterType;
  end
This code is used in section 1898.
         \langle Parse existential registration cluster 1903\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;
  gSubexpPtr \uparrow. Complete Type; \ gSubexpPtr \uparrow. Complete Cluster Type; \ KillSubexpression; \ KillExpression;
  gItemPtr\uparrow.FinishClusterType;
  end
This code is used in section 1898.
1904. Reduction registration.
\langle \text{ Parse definitions (parser.pas) } 1868 \rangle + \equiv
procedure Reduction;
  var lExpKind: ExpKind;
  begin gBlockPtr↑.CreateItem(itReduction); ReadWord;
  if (CurWord.Kind = Identifier) \land (AheadWord.Kind = sy\_Arrow) then ErrImm(paFunctExp4);
  qItemPtr \uparrow. StartFuncReduction; TermExpression; qItemPtr \uparrow. ProcessFuncReduction;
  Accept(sy\_To, paToExp); TermExpression; gItemPtr \uparrow. FinishFuncReduction; Semicolon; Correctness;
  end;
         \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paFunctExp4 = 302; paToExp = 404;
1906. Identification registration.
\langle \text{Parse definitions (parser.pas) } 1868 \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 qItemPtr \uparrow .ProcessLeftLocus; Accept(Identifier, paIdentExp3);
       Accept(sy\_Equal, paEqualityExp1); gItemPtr\uparrow.ProcessRightLocus; Accept(Identifier, paIdentExp3);
     until \neg Occurs(sy\_Comma);
     end;
  Semicolon; Correctness;
  end:
```

580 DEFINITIONS Mizar Parser §1907

```
1907. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paWithExp = 390;
1908. Property registration.
\langle \text{Parse definitions (parser.pas) } 1868 \rangle + \equiv
procedure RegisterProperty;
  begin gBlockPtr\uparrow.CreateItem(itPropertyRegistration);
  case PropertyKind(CurWord.Nr) of
  sySethood: begin ReadWord; Accept(sy\_of, paOfExp); gItemPtr\uparrow.StartSethoodProperties;
     TypeExpression; gItemPtr \uparrow . FinishSethoodProperties; Justification;
     end;
  othercases begin SynErr(CurPos, paStillNotImplemented);
     end;
  endcases;
  Semicolon;
  end:
         \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paStillNotImplemented = 400;
1910.
\langle \text{Parse definitions (parser.pas) } 1868 \rangle + \equiv
procedure RegistrationBlock;
  var DefPos: Position;
  begin gBlockPtr\uparrow.CreateItem(itDefinition); gBlockPtr\uparrow.CreateBlock(blRegistration);
  DefPos \leftarrow CurPos; ReadWord;
  while CurWord.Kind \neq sy\_End do
    \mathbf{begin} \ \mathit{StillCorrect} \leftarrow \mathit{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} \ gBlockPtr\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;
```

§1911 Mizar Parser DEFINITIONS 581

```
1911.
         Reservation.
\langle \text{Parse definitions (parser.pas) } 1868 \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;
1912.
         \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paIdentExp11 = 300;
1913. Theorem.
\langle \text{Parse definitions (parser.pas) } 1868 \rangle + \equiv
procedure Theorem;
  begin gBlockPtr\uparrow.CreateItem(itTheorem); ReadWord; ProcessLab; <math>gItemPtr\uparrow.StartTheoremBody;
  ProcessSentence;\ gItemPtr \uparrow. FinishTheoremBody;\ Justification;\ gItemPtr \uparrow. FinishTheorem;
  end;
1914. Axiom.
\langle \text{Parse definitions (parser.pas) } 1868 \rangle + \equiv
procedure Axiom;
  begin qBlockPtr\uparrow.CreateItem(itAxiom); ReadWord; ProcessLab; <math>qItemPtr\uparrow.StartTheoremBody;
  ProcessSentence;\ gItemPtr\uparrow.FinishTheoremBody;\ gItemPtr\uparrow.FinishTheorem;
  end;
```

582 SCHEME BLOCKS Mizar Parser $\S1915$

Section 24.8. SCHEME BLOCKS

```
1915.
\langle \text{Parse scheme block (parser.pas)} 1915 \rangle \equiv
    { Main (with Schemes) }
procedure SchemeBlock;
  var SchemePos: Position;
  begin qBlockPtr\uparrow.CreateItem(itSchemeBlock); qBlockPtr\uparrow.CreateBlock(blPublicScheme); ReadWord;
  gBlockPtr\uparrow.CreateItem(itSchemeHead);\ gItemPtr\uparrow.ProcessSchemeName;\ SchemePos \leftarrow PrevPos;
  if CurWord.Kind = Identifier then ReadWord;
  (Parse scheme parameters 1917);
  Accept(sy\_RightCurlyBracket, paRightCurledExp3); gItemPtr\uparrow.FinishSchemeHeading;
  Accept(sy\_Colon, paColonExp3); FormulaExpression; { Scheme-conclusion }
  gItemPtr\uparrow.FinishSchemeThesis; \langle Parse scheme premises 1919 \rangle;
  gItemPtr\uparrow.FinishSchemeDeclaration; \langle Parse justification for scheme 1920 \rangle;
  KillBlock:
  end;
This code is used in section 1673.
1916. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paRightCurledExp3 = 372; paColonExp3 = 384;
1917. \langle Parse scheme parameters 1917\rangle \equiv
  Accept(sy\_LeftCurlyBracket, paLeftCurledExp);
  repeat qItemPtr\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;
       end;
    othercases begin ErrImm(paWrongSchemeVarQual); gItemPtr \uparrow . FinishSchemeQualification;
       Specification;
       end:
    endcases; gItemPtr \uparrow . FinishSchemeSegment;
  until \neg Occurs(sy\_Comma)
This code is used in section 1915.
1918. \langle \text{Error codes for parser } 1675 \rangle + \equiv
  paIdentExp13 = 300; paLeftCurledExp = 362; paWrongSchemeVarQual = 364;
  paRightParenthExp9 = 370; paRightSquareExp5 = 371;
1919. \langle Parse scheme premises 1919\rangle \equiv
  if CurWord.Kind = sy\_Provided then
    repeat gItemPtr\uparrow.StartSchemePremise; ReadWord; ProcessLab; ProcessSentence;
       gItemPtr\uparrow.FinishSchemePremise;
    until CurWord.Kind \neq sy\_And
This code is used in section 1915.
```

 $\S1920$ Mizar Parser SCHEME BLOCKS 583

```
1920.
        \langle \text{ Parse justification for scheme } 1920 \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 1915.

584 MAIN PARSE PROCEDURE Mizar Parser §1921

Section 24.9. MAIN PARSE PROCEDURE

1921. 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
\( \text{Main parse method (parser.pas) 1921} \) \geq \quad \text{procedure } Parse;
\text{begin } skip\_to\_begin; \quad \text{Skips ahead until EOT or finds 'begin'} \} \text{if } CurWord.Kind = EOT \text{ then } ErrImm(213) \text{else } \left{ Parse proper text } 1922 \right{\gamma}; \quad \text{CurrWord.Kind} = sy\_Begin \right{ KillBlock; \text{end;}} \text{end;} \]

This code is used in section 1673.
```

1922. 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 (§1394) and StartProperText (§1396) are both implemented in the extended block class.

```
[The 213 magic number should be made a constant, something like paBegExpected?] \langle Parse proper text 1922 \rangle \equiv begin gBlockPtr \uparrow .StartProperText; <math>gBlockPtr \uparrow .ProcessBegin; Accept(sy\_Begin, 213); while CurWord.Kind \neq EOT do \langle Parse next block 1923 \rangle; end
```

This code is used in section 1921.

§1923 585 Mizar Parser MAIN PARSE PROCEDURE

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

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 1923} \rangle \equiv
  begin (Parse pragmas and begins 1924);
  StillCorrect \leftarrow true;  { we are not in panic mode }
  if CurWord.Kind = sy\_End then
    begin (Skip all end tokens, report errors 1925);
    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 1922.

The ProcessPragmas (§1815) 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 } 1924 \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 1923.

1925. 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 1925\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 1923.
```

1926. 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.

```
¡Error, 391: 1805.
                                                          AcceptEndState: 661, 670, 836, 837.
CHReport: 14.
                                                          AcceptStartState: \underline{661}, \underline{670}.
.frt file: 1030.
                                                          Accomodation: 102, 103.
                                                          AChar: 693, 1328.
.idx file: 893.
.prf File: 887.
                                                          aChar: 43, 45, 698, 1330.
.wsx file: 1030.
                                                          aCluster: 964, 965, 1010, 1015, 1194, 1200,
::$EOF: 774.
                                                              1328, 1336.
a: 201.
                                                          aCode: 322, 336, 579, 580, 597, 598.
                                                          a Comment: 108, 109.
aAdjectiveNr: 911, 912.
aAdjectives: 976, 977.
                                                          aCompare: 387, 388, 398, 399, 400, 409, 410,
aAncestors: 1110, 1111.
                                                              428, 429.
a Another \colon \ \ \underline{322}, \, \underline{326}, \, \underline{327}, \, \underline{334}, \, \underline{349}, \, 365, \, \underline{368}, \, \underline{369},
                                                          aConcl: 1068, 1069.
    <u>387</u>, <u>389</u>, 390, <u>452</u>, <u>454</u>.
                                                          aCond: 646, 651, 1328, 1361, 1362.
AAnother: \underline{349}, \underline{365}.
                                                          aConds: 1086, 1087, 1162, 1163.
aAntec: 1171, 1172.
                                                          aCons: 1167, 1168, 1169, 1170, 1171, 1172,
aApplicationName: 110, 111.
                                                              1173, 1174.
aArg: 909, 910, 930, 931, 932, 933, 938, 939,
                                                          aCStm: 1194, 1255, 1328, 1352.
    982, 983, 1117, 1118.
                                                          aDctFileName: 753, 756, 759, 760, 792, 805,
aArgs: 911, 912, 924, 925, 926, 927, 928, 929, 936,
                                                              883, 890, 892.
    937, 958, 959, 960, 961, 962, 963, 978, 979,
                                                          Add: 211, 223, 238, 239, 240, 244, 245, 253, 261,
                                                              528, 544, 579, 595.
    1115, 1116, 1117, 1118, 1121, 1122.
aArgsNbr: 817, 818, 825, 826, 828, 831, 833.
                                                          AddExtItems: 370, 384, 409, 422.
aArticleExt: 1036, 1037.
                                                          AddExtObject: 370, 383, 409, 416, 421, 422.
aArticleID: 1036, 1037.
                                                          AddObject: 452, 460, 461, 743.
aArticleNr: 1051, 1052, 1061, 1062.
                                                          addr: 443, 447.
aAttr: 1010, 1014, 1194, 1199, 1328, 1335.
                                                          Address of: 320.
aAttrName: 688, 690, 691, 692, 1277, 1279.
                                                          AddSequence: \underline{552}, 555, 559.
aBase: 28, 1379, 1573.
                                                          AddString: 452, 457, 458, 460, 472.
                                                          AddStrings: 452, 461.
aBlock: 1194, 1254, 1255, 1256, 1328, 1352,
    1354, 1360.
                                                          aDef: 1130, 1131, 1140, 1141, 1142, 1143, 1144,
                                                              1145, 1146, 1147, 1194, 1248, 1328, 1366.
aBlockKind: 1036, 1038.
aBlokKind: 1036, 1039.
                                                          aDefKind: 1136, 1137.
Abs: 204, 205, 269.
                                                          a DefNr: 1051, 1052.
                                                          aDefWay: 1146, 1147.
abs: 208, 235, 237, 240, 241, 242, 244, 245,
    246, 249, 251, 255.
                                                          aDelta: 429, 506, 507.
abstract\_syntax\colon \quad \underline{895},\ 1030,\ 1379.
                                                          ADelta: 349, 350, 428, 429, 507, 528, 529, 547, 548.
Abstract1: 308, 430, 450, 611, 788.
                                                          aDir: 53, 54.
AbsVocabularyObj: 731, 732, 733.
                                                          AdjectiveExpressionObj: 907, 908, 909, 911.
Abs Vocabulary Ptr: \quad {\bf 731}.
                                                          AdjectiveExpressionPtr: 907, 909, 910, 1010, 1014,
aCapacity: 452, 454, 480, 552, 554, 567.
                                                              1194, 1199, 1277, 1288, 1328, 1335, 1574.
Accept: 1683, 1688, 1693, 1696, 1698, 1700, 1703,
                                                          AdjectiveObj: 911, 912.
    1705, 1709, 1711, 1714, 1719, 1722, 1734, 1740,
                                                          AdjectivePtr: 911, 1014, 1199, 1288, 1335, 1575.
    1742, 1749, 1757, 1762, 1764, 1767, 1769, 1783,
                                                          AdjectiveSort: 907, 908.
    1785, 1787, 1794, 1796, 1798, 1807, 1809, 1811,
                                                          Adjective SortName: 1288.
    1813, 1823, 1835, 1843, 1845, 1849, 1851, 1856,
                                                          aElem: 646, 652.
    1859, 1861, 1864, 1866, 1873, 1875, 1877, 1879,
                                                          aEnvname: 66, 68.
    1881, 1883, 1900, 1902, 1904, 1906, 1908, 1911,
                                                          aEnvName: 65, 66.
    1915, 1917, 1920, 1922.
                                                          aEqList: 1177, 1178.
AcceptEnd: 1680, 1802, 1822, 1830, 1887, 1892,
                                                          aErr: 646, 652, 661, 664.
    1910, 1920.
                                                          aErrorNbr: 125, 126.
```

aExp: 1138, 1139. 501, 552, 561, 563, 565, 566, 568, 571, 573, <u>579</u>, 584, <u>587</u>, <u>597</u>, 602, <u>6</u>05. aExpKind: 28, 1010, 1011, 1895, 1897. aInf: 1194, 1252, 1253, 1254, 1328, 1357, 1358, aExpr: 1128, 1129.1359, 1360. aExt: 53, 54, 192, 193, 194, 195. aInferSort: 1055, 1056, 1057, 1058. aFields: 1106, 1107, 1110, 1111. aInfo: 322, 336, 579, 580, 597, 598. aFieldSymbNr: 1104, 1105. aInt: 43, 46, 552, 558, 564, 565, 566, 568, 569, aFileExt: 63, 64, 72, <u>74</u>, 75. 570, 571, 572, 573, 574. aFileName: 47, 48, 51, 52, 53, 54, 55, 56, 57, 58, AItem: 370, 378, 409, 418. 59, 60, 61, 62, 63, 64, 70, 73, 74, 653, 654, 655, aItem: 322, 332, 333, 337, 349, 362, 370, 373, 387, 661, 662, 717, 733, 736, 740, 743, 744, 745, 391, 392, 393, 405, 409, 416, 428, 433, 434, 792, 805, 883, 890, 892, 1194, 1195, 1276, 1277, 486, 490, 497, 499, 500, 506, 510, 579, 583, 1278, 1327, 1328, 1329, 1372. 597, 601, 1277, 1322, 1323. AFileName: 680, 682, 684, 686, 688, 689, 693, aItemKind: 1036, 1038, 1040, 1041. 694, 695, 1329. aItems: 492, 493, 585. AFilename: 694. AItems: 492, 493, 585. aFormat: 825, 830. aItem1: 387, 807, 819, 820, 821, 822, 823, 824. aFormNr: 811, 838. aItem2: 387, 807, 819, 820, 822, 823, 824. *aFormula*: 944, 945. aIters: 1096, 1098. *aFrm*: <u>1010</u>, 1022, 1023, 1024, 1025, <u>1027</u>, 1194, aJustification: 1080, 1081, 1086, 1087, 1093, 1094, 1206, 1207, 1208, 1209, 1210, 1211, 1212, 1213, 1095, 1096, 1097, 1098, 1158, 1159, 1160, 1161, 1214, 1215, 1216, 1217, 1218, 1219, 1220, 1221, 1162, 1163, 1164, 1165. 1328, 1343, 1344, 1345. ak: 1606, 1608. AfterBalance: 1594, 1597, 1598, 1599, 1600, aKey: 387, 401, 402, 403, 404, 409, 412, 413, 414. 1603, 1604, 1605. aKey1: 424, 540, 611. aFunc: 497, 502, 528, 544, 545, 579, 588, 595, aKey2: 424, 540, 611. <u>597</u>, <u>606</u>. aKind: 751, 752, 761, 762, 811, 812, 813, 814, 815, aFuncId: 1073, 1074.816, 825, 826, 833, 924, 925, 930, 931, 958, 959,aFunctorIdNr: 928, 929. 1080, 1081, 1121, 1122, 1126, 1127, 1128, 1129, aFunctorNr: 922, 923. 1167, 1168, 1179, 1180, 1181, 1182, <u>1373</u>, <u>1374</u>. aFunc1: 528, 530. alab: 1045. aFunc2: 528, 530. aLab: 1044, 1045, 1091, 1092, 1126, 1127, 1130, Aggregate TermObj: 936, 937. 1131, 1134, 1135, 1194, 1247, 1328, 1350. Aggregate TermPtr: 936, 1029, 1229, 1301, 1348, aLabelId: 1044, 1045.1625. aLabId: 1048, 1049.aGuard: 1132, 1133. aLArgs: 1119, 1120, 1121, 1122. $ahead_is_attribute_argument$: 1725. aLArgsNbr: 815, 816, 817, 818, 825, 827, 828, $ahead_is_type: \underline{1895}.$ 829, 831, 832, 834. AheadPos: 883, 886, 889. aLBSymb: 1121, 1122.AheadWord: 883, 886, 889, 1417, 1419, 1542, 1545, aLeft: 409, 414. 1548, 1588, 1626, 1628, 1696, 1709, 1711, 1725, aLeftArg: 984, 985, 986, 987, 988, 989, 990, 991, 1758, 1760, 1766, 1767, 1768, 1769, 1774, 1794, 992, 993, 994, 995, 996, 997. 1818, 1820, 1840, 1880, 1895, 1897, 1898, 1904. aLeftArgs: 922, 923, 972, 973. aID: 79. aLeftBracketNr: 926, 927.aIdentifier: 903, 904. aLeftLocus: 1175, 1176.aIdentifiers: 905, 906, 1070, 1071. a Length: 36, 38.aIdentNr: 898, 899, 916, 917, 1108, 1109. aLexem: 753, 755.aIdNr: 1068, 1069, 1193. aLimit: 324, 325, 370, 387, 388, 394, 395, 396, aIndex: 322, 333, 335, 341, 370, 381, 382, 387, 409, 410, 419, 429, 486, 488, 492, 497, 498, 506, 391, 398, 401, 402, 403, 404, 408, 409, 412, 507, 508, 579, 581, 585, 597, 599, 603, 611. <u>413</u>, 415, 420, <u>452</u>, 453, 463, <u>465</u>, 467, 468, ALimit: 322, 343, 344, 345, 349, 350, 370, 371. 472, 473, 475, 477, 478, 486, 491, 497, 500, 379, 380, 387, 388, 409, 410, 419, 423, 425,

428, 429, 492, 493, 507, 528, 529, 547, 548, 1058, 1059, 1060, 1061, 1062, 1064, 1065, 1066, 585, 825, 835. 1067, 1068, 1069, 1095, 1097, 1104, 1105, 1106, aLine: 718, 722, 724, 1373, 1376, 1377. 1107, 1108, 1109, 1110, 1111, 1113, 1114, 1115, aLinked: 1059, 1060.1116, 1117, 1118, 1119, 1120, 1121, 1122, 1126, 1127, 1130, 1131, 1134, 1135, 1136, 1137, 1138, aLinkPos: 1059, 1060.aList: 398, 553. 1139, 1140, 1141, 1142, 1143, 1144, 1145, 1146, 1147, 1148, 1149, 1150, 1151, 1152, 1153, 1154, Allowed: 79, 792, 804, 805, 886. 1155, 1156, 1157, 1158, 1159, 1160, 1161, 1162, aLoci: 1194, 1240, 1328, 1364. 1163, 1164, 1165, 1167, 1168, 1169, 1170, 1171, aLocus: 1194, 1239, 1328, 1363. 1172, 1173, 1174, 1175, 1176, 1177, 1178, 1179, aLocusNr: 918, 919. 1180, 1181, 1182, 1183, 1184. aLSymNr: 817, 818, 825, 828, 831. Apos: 933. aMmlVcb: <u>717</u>, <u>743</u>, 744, 745. aPostqual: 942, 943, 944, 945. aModeSymbol: 960, 961.append: 140.AnalyzerOnly: 93, 105. $append_dir_separator: 643, 644.$ aName: 53, 54, 115, 116, 649, 650, 1185, 1186. AppendErrors: 139, $\underline{140}$. aName1: 49, 50.AppendFunc: 1714, 1718, 1720, 1722, 1754. aName2: 49, 50. $AppendQua: \quad \underline{1689}, \ 1713.$ Anaphora: 952. AppendTo: 322, 346.aNbr: 1373, 1374.aPredId: 1075, 1076.aNewPatt: 1148, 1149, 1177, 1178. aPredIdNr: 978, 979. aNewTerm: 1183, 1184. aPredNr: 970, 971, 972, 973. aNr: 751, 752, 761, 762, 1061, 1062. aPredSymbol: 1746, 1751. AnsiChar: 698, 756, 759, 1188. aPrems: 1068, 1069. aNSort: 1148, 1149. aPrfFileName: 883, 887. Antonym: 1883, 1894. aPrg: 1373, 1374, 1375, 1376. aNumber: 1328, 1332. aProgName: 192, 193.aObject: 452, 460, 474, 475, 478. aProp: 1093, 1094, 1096, 1098, 1152, 1153, 1194, aOrigPatt: 1148, 1149, 1177, 1178. 1249, 1328, 1351. aOrigTerm: 1183, 1184. aProps: 1154, 1155, 1156, 1157. aOtherwise: 1134, 1135. aQualVars: 1086, 1087. aOverArgs: 1110, 1111. aQVars: 1156, 1157.aParams: 1068, 1069. aRArgs: 1119, 1120, 1121, 1122. aParenthCnt: 1752, 1754.aRArgsNbr: 813, 814, 815, 816, 817, 818, 825, aPartDef: 1132, 1133.827, 828, 829, 831, 832, 834. aPartialDefs: 1134, 1135. aRBSymb: 1121, 1122. aPassName: 186, 187.aRedef: 1136, 1137, 1140, 1141, 1142, 1143, 1144, aPattern: 1136, 1137, 1138, 1139, 1140, 1141, 1145, 1146, 1147. 1142, 1143, 1144, 1145, 1146, 1147, 1194, 1241, AreEqComplex: 280, 281.1242, 1243, 1244, 1245, 1246, 1328, 1365. aRef: 1194, 1250, 1328, 1355. aPos: 761, 762, 898, 899, 901, 902, 903, 904, 905, aRefs: 1194, 1251, 1328, 1356. 906, 907, 908, 909, 910, 911, 912, 916, 917, 918, aRes: 1194, 1257. 919, 920, 921, 922, 923, 924, 925, 926, 927, 928, aResType: 1328, 1369.929, 930, 931, 932, 933, 934, 935, 936, 937, 938, arg_type: 1567, 1568. 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, ArgsLength: 1597, 1598, 1600, 1601, 1602, 1603,949, 950, 951, 952, 953, 954, 955, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967, 970, 971, 972, 1604, 1605. Arguments Tail: 1691, 1705, 1718, 1722, 1737, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 986, 987, 988, 989, 990, 991, 1738, 1754. 992, 993, 994, 995, 996, 997, 998, 999, 1000, aRight: 409, 414. 1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, aRightArg: 984, 985, 986, 987, 988, 989, 990, 991, 1009, 1036, 1037, 1038, 1039, 1040, 1041, 1044, 992, 993, 994, 995, 996, 997. 1045, 1048, 1049, 1051, 1052, 1055, 1056, 1057, aRightArgs: 922, 923, 970, 971, 972, 973.

aRightBracketNr: 926, 927. AString: 693.aRightLocus: 1175, 1176.aString: 36, 38, 39, 40, 41, 42, 701, 1328, 1333. aRStm: 1194, 1256, 1328, 1354. aStrings: 452, 461. aRSymNr: 817, 818, 825, 828, 831. aStructSymb: 1110, 1111. ArticleExt: 36, 75, 105, 643, 1383. aStructSymbol: 936, 937, 938, 939, 940, 941, ArticleID: 36, 75, 760, 1383. 962, 963. aStr1: 438, 611.ArticleName: 36, 75, 643. aSample: 942, 943, 944, 945. aStr2: 438, 611. aSch: 1194, 1258, 1328, 1370. aSubject: 946, 947, 948, 949, 976, 977, 980, 981. AsciiArr: 750.aSym: 661, 664. aSymb: 1115, 1116, 1117, 1118, 1119, 1120, ASCIIArr: 749, 792, 793. aScope: 998, 999, 1000, 1001, 1002, 1003. 1121, 1122. aScraps: 974, 975. aSymNr: 811, 812, 813, 814, 815, 816, 825, 826, aSegm: 1010, 1017, 1018, 1194, 1204, 1205, 827, 829, 832, 833, 834. 1328, 1338, 1339. At: 322, 330, 331, 355. aSegment: 998, 999, 1000, 1001, 1002, 1003. atAid: 615, 760.aSegmSort: 1064, 1065. atArgNr: 615, 837, 838. aSelectorNr: 932, 933, 934, 935. atArticleExt: 615, 1196, 1324. aSentence: 1044, 1045. atArticleID: 1324. aSeq: 552, 555, 556, 559.atArticleId: 615, 1196. aSet: 568, 575, 576, 577, 578. atCol: 615, 712, 1280, 1324, 1325, 1326. aShape: <u>1277</u>, 1321, <u>1322</u>, 1323. atCondition: 615, 1259, 1267, 1322, 1323. aSiq: 160.atConstrNr: 615. aSignal: 165, 166. AtDelete: 349, 352, 355, 360, 363, 486, 491, 517, aSnt: 1075, 1076. 519, 534, 552, 561, 570, 579, 584, 593, 597, 602. aTerm: 1073, 1074, 1077, 1078, 1080, 1081, 1083, aSntPos: 1044, 1045.aSort: 901, 902, 907, 908, 1113, 1114, 1150, 1151, 1084, 1095, 1097. 1160, 1161, 1164, 1165. aText: 722, 728, 729, 733, 738, 739, 740. aSpec: 1106, 1107, 1140, 1141, 1146, 1147. AtFree: 349, 355, 790. aSpecification: 1066, 1067. a Theo Nr: 1051, 1052. $aSpelling \colon \ \, 751, \, 752, \, 753, \, 755, \, 761, \, 762.$ atIdNr: 615, 1202, 1206, 1222, 1223, 1239, 1247, Assert: 1767, 1769. 1250, 1253, 1258, 1291, 1294, 1298, 1299, 1303, assert_valid_index: 463, 464, 467, 468, 472, 1311, 1313, 1316, 1319. 477, 478. aTime: 115, 116, <u>170</u>. AtIndex: 409, 415. $assert_valid_index_for_nat_set$: 491, 499, 500, 510. assign: 756, 759, 805, 887, 1189, 1192. atInfinitive: 615, 747. Assign: 70, 135, 140, 152, 528, 530, 531, 579, 591, AtInsert: 322, 333, 349, 356, 362, 387, 391, 392, 595, 597, 609, 625, 626, 630, 632, 635, 640, 655, 434, 435, <u>497</u>, 499, 500, 544, <u>552</u>, 565, <u>568</u>, 573. 682, 686, 736, 743, 744, 836, 1377. atKind: 615, 746, 747, 760, 836, 837, 838, 839, $assign_gcd_and_jump$: 234, 235. 1197, 1248, 1275, 1310, 1325, 1326. Assigned: 412.atLabelNr: 615. atLeftArqNr: 615, 837, 838. AssignPair: 486, 496, 497, 505. Assumption: 1779, 1817, 1822, 1891. atLine: 615, 712, 1280, 1324, 1325, 1326. AssumptionKind: 1150, 1151.atMizfiles: 615, 760. AssumptionKindName: 1272, 1323.atName: 615, 745, 747, 760. AssumptionObj: 1150, 1151, 1152, 1154, 1157. atNegated: 615.AssumptionPtr: 1150, 1272, 1328, 1362, 1371. atNr: 615, 746, 760, 838, 1199, 1201, 1215, 1216, aStatementSort: 1089, 1090, 1091, 1092. 1224, 1226, 1227, 1228, 1229, 1230, 1231, 1232, aStr: 43, 44, 45, 110, 111, 317, 318, 423, 426, 1242, 1243, 1244, 1245, 1246, 1251, 1253, 1271, 1288, 1290, 1297, 1300, 1301, 1305, 1306, 1307, 427, 452, 453, 457, 458, 460, 465, 470, 471, 472, 473, 475, 477, 674, 676, 679, 1042, 1043, 1308, 1314, 1319, 1323. 1283, 1284, 1285, 1286. atNumber: 615, 1225, 1251, 1301, 1314.

aToken: 764, 765, 766, 767. aTypeExpList: 1064, 1065, 1066, 1067, 1073, 1074, 1075, 1076. atOrigin: 615.aTypeExpr: 1080, 1081. atPosCol: 615, 1197, 1253, 1275, 1319, 1325, 1326. atPosLine: 615, 1197, 1253, 1275, 1319, 1325, aTypeList: 1194, 1238, 1328, 1349. 1326. at Y1: 615.atPriority: 615, 747. at Y2: 615.atProperty: 615, 1259, 1322. AvailableSymbols: 715, 719, 739, 740, 746. AtPut: 349, 359, <u>552</u>, 566. aVal: 661, 673, 688, 690. atRightSymbolNr: 615, 837, 838. a Value: 452, 481, 649, 650, 920, 921. aTrm: 1010, 1026, 1028, 1029, 1173, 1174, 1194, aVar: 1010, 1016, 1080, 1081, 1194, 1202, 1203, 1222, 1223, 1224, 1225, 1226, 1227, 1228, 1229, 1328, 1337. 1230, 1231, 1232, 1233, 1234, 1235, 1236, 1237, aVarId: 1077, 1078, 1083, 1084. 1328, 1346, 1347, 1348. a Vars: 1064, 1065, 1066, 1067. aTrmList: 1010, 1020, 1194, 1198, 1328, 1340, aWSBlock: 1194, 1197, 1328, 1367. 1341. aWSItem: 1194, 1259, 1260, 1275, 1328, 1371. atSchNr: 615. aWsItem: 1275.atSerialNr: 615. aWSTextProper: 1194, 1196, 1276, 1328, 1368, atShape: 615, 1248, 1259, 1310, 1322. 1372.atSpelling: 615, 1199, 1201, 1202, 1206, 1215, Axiom: 1914, 1923. 1216, 1222, 1223, 1224, 1226, 1227, 1228, 1229, Axioms: 104. 1230, 1231, 1239, 1242, 1243, 1244, 1245, 1246, AxiomsAllowed: 93, 105, 1417, 1433, 1543. 1247, 1250, 1251, 1253, 1258, 1259, 1271, 1322. *aXMLStr*: 674, 677. atSymbolNr: 615, 836, 837, 838, 839. a1: 201, 203, 211, 212, 213, 214, 215, 216, 217, AttributeDefinitionObj: 1142, 1143. 218, 221, 222, 223, 235, 237. AttributeDefinitionPtr: 1142, 1269, 1323, 1371, $b_{-}GPC$: 226, 227. 1446. baseunix: 154.AttributeName: 1187, 1189, 1190, 1191, 1199, begin: 11. 1246, 1335, 1365. BI: 648, 659, 662. Attribute Pattern Obj: 1117, 1118.BiconditionalFormulaObj: 992, 993. AttributePatternPtr: 1117, 1142, 1143, 1246, BiconditionalFormulaPtr: 992, 1296, 1658. 1277, 1306, 1323, 1365, 1446, 1491. BinaryArgumentsFormula: 984, 985, 986, 988, AttributeSymbol: 884, 1491, 1575, 1706, 1725, 990, 992, 994, 996. 1758, 1845, 1880, 1881, 1895. BinaryFormulaPtr: 984, 1010, 1022, 1025, 1209, Attributesymbol: 1883. 1210, 1211, 1212, 1213, 1214, 1328, 1343, 1345. AttributiveFormulaObj: 976, 977. BinIntFunc: 579, 580, 581, 582, 583, 584, 585, AttributiveFormulaPtr: 976, 1025, 1218, 1295, <u>586, 587, 588, 589, 590, 591, 592, 593, 594,</u> 1345, 1647. 595, 596, 808. AttrLookup Table: 1281, 1282, 1284. BinIntFuncError: <u>579</u>, 580, 583, 584, 591, ATTSubexpression: <u>1895</u>, 1898, 1900, 1902. 593. 594. atUnknown: 615, 1284. BinIntFuncPtr: 579.at Value: 615, 836, 839. biStackedObj: 1010.at VarNr: 615. biStackedPtr: 1010, 1013.at Var Sort: 615. Bl: 1597, 1598, 1600, 1603, 1604, 1605. atX: 615.blCase: 852, 1367, 1391, 1828. at X1: 615.blDefinition: 852, 1367, 1391, 1887. at X2: 615.blDiffuse: 852, 1367, 1391, 1830. blHereby: 852, 1367, 1391, 1830. at Y: 615.aTyp: 1010, 1021, 1167, 1168, 1169, 1170, 1171, blMain: 852, 1037, 1196, 1285, 1324, 1382, 1172, 1173, 1174, 1194, 1201, 1328, 1342. 1383, 1386. aType: 905, 906, 946, 947, 950, 951, 964, 965, 980, blNotation: 852, 1367, 1391, 1892. 981, 1070, 1071, 1181, 1182. BlockKind: 852, 853, 854, 859, 1035, 1036, 1038, $1039,\ 1281,\ 1285,\ 1385,\ 1386,\ 1403.$ aTypeChangeList: 1080, 1081.

BlockLookup Table: 1281, 1282, 1285. CheckTermLimit: 1584, 1585. BlockLookUpTable: 1281. checkzero: 201, 206, 208, 209, 210, 212, 215, 222, BlockName: 1035, 1196, 1197, 1281, 1324. 224, 226, 233, 235, 237, 239, 244, 249, 251. BlockObj: 853, 854, 855, 856, 857, 858, 859, Choice operator: 950, 1237. 860, 1385. Choice, Axiom of: 950. BlockPtr: 850, 853, 856, 859. $choice_operator_cases$: 1706. BlockWrite: 696, 699. ChoiceStatementObj: 1086, 1087. blProof: 852, 1367, 1391, 1393, 1830. ChoiceStatementPtr: 1086, 1262, 1323, 1371, blPublicScheme: <u>852</u>, 1367, 1391, 1915. 1437. blRegistration: 852, 1367, 1391, 1910. Choice TermObj: 950, 951.blSuppose: 852, 1367, 1391, 1825, 1828. ChoiceTermPtr: 950, 1029, 1237, 1301, 1348, BOOL: 166. 1631. boolean: 128, 133, 206, 254, 272, 273, 274, 275, Chr: 45. $chr\colon\ 79,\ 657,\ 678,\ 749,\ 753,\ 758,\ 770,\ 839.$ 279, 688, 690, 718, 753, 755, 764, 788, 791, 792, 803, 804, 911, 1059, 1060, 1136, 1137, 1140, CHReport: 287, 289, 291, 294, 297. 1141, 1142, 1143, 1144, 1145, 1146, 1147, 1194, CIm Unit: 277.1277, 1318, 1323, 1328, 1345, 1348, 1377, 1385, CircumfixFunctor: 1121, 1122, 1241, 1244, 1307, 1405, 1408, 1412, 1423, 1425, 1472, 1554, 1568, 1309. 1365. 1597, 1672, 1683, 1870, 1895. CircumfixFunctorPatternObj: 1121, 1122.borrow: 216, 218. CircumfixFunctorPatternPtr: 1121, 1244, 1307, 1365, 1503. Borrow: $\underline{216}$. Bourbaki, Nicolas: 1101, 1102. CircumfixTermObj: 926, 927. Bracket: 926. CircumfixTermPtr: 926, 1029, 1228, 1301, 1348, 1623. BracketedTerm: 1690, 1700.break: 69, 105, 200, 253, 337, 403, 666, 669, Clear: 452, 462, 552, 560. 779, 782, 793, 1377, 1607, 1888, 1893, 1895, Close: 626, 683, 736, 743, 744. 1910, 1923. close: 70, 142, 144, 152, 625, 639, 656, 756, 759, BuffChar: 680.806, 887, 1189, 1192, 1329. Bug: 204. ClosedParenth: 1688.Bug, Memory leak: 805. $CloseEmptyElementTag: \underline{661}, 669, 672.$ BugInProcessor: 120, 121, 155, 156. CloseErrors: 143, 144, 155. builtin: 198. CloseInfoFile: 152. $Close Infofile \colon \ \underline{151}, \ \underline{152}.$ Byte: 740. *b1*: 201, 203, 211, 212, 213, 214, 215, 217, 218, CloseParenth: <u>1688</u>, 1713, 1764. 221, 222, 223, 235, 237. CloseSourceFile: 883, 891, 893. c: 211, 219, 676. CloseStartTag: <u>661</u>, 669, 672. Canceled: 1790, 1891, 1910, 1923. Clustered Type Obj: 964, 965. CanceledPragma: 1373, 1374.Clustered Type Ptr: 964, 1021, 1201, 1290, 1342, CapitalizeName: 1185, 1186. 1591, 1646. CasePos: 1821, 1822, 1825, 1828, 1829. ClusterObj: 1167, <u>1168</u>, 1169, 1170, 1171, 1172, Catch Signal: 160.1173, 1174. cchBuffer: 67, 69. ClusterPtr: 1167, 1273, 1371. CClusterObj: 1171, 1172.ClusterRegistrationKind: 1167, 1168, 1408. CClusterPtr: 1171, 1273, 1323, 1371, 1452. ClusterRegistrationName: 1273, 1323. cdecl: 160, 166. ClustersProcessing: 92, 96, 98, 99, 100. ch: 764, 788, 791, 792, 804. CMinusOne: 277.CH_REPORT: 198, 287, 289, 291, 294, 297. cmSecs: 169, 170, 171, 175, 177, 181. ChangeFileExt: 63, 64, 73, 74. cmt: 792, 802. Char: 738, 740, 811, 812, 813, 814, 815, 816. coConsistentError: 310, 534, 593. CharKind: 657, 659. Code: 453, 466, 486, 487, 552, 562, 680, 681, CheckCompatibility: 617, 639. 684, 685. CheckerOnly: 93, 105. coDuplicate: 310, 459, 535, 591, 594, 601, 610.

coIndexError: 310, 330, 333, 335, 353, 463, 491, 561, 563, 565, 566, 583, 584, 601, 602. coIndexExtError: 310, 374, 383, 410, 412, 415, 417, 420, 421. Col: 128, 129, 132, 133, 138, 151, 654, 657, 712, 768, 773, 774, 777, 782, 787, 789, 793, 795, 796, 800, 803, 889, 1197, 1253, 1275, 1280, 1319, 1324, 1325, 1326, 1679. CollectBracketForm: 825, 831, 1503. CollectFormat: 825, 830. CollectFuncForm: <u>825</u>, 832, 1503. Collective Assumption: 1150, 1155, 1157, 1272, 1323, 1362. CollectiveAssumptionObj: 1154, 1155, 1156. CollectiveAssumptionPtr: 1154, 1272, 1323, 1362, 1439, 1443. CollectPredForm: 825, 834, 1499. CollectPrefixForm: 825, 833, 1491, 1494, 1507, 1509, 1510, 1513. Collect Token: 753, 755.CompactStatementObj: 1093, 1094, 1096. CompactStatementPtr: 1093, 1194, 1255, 1256, 1260, 1277, 1317, 1321, 1328, 1352, 1354, 1371, 1400, 1430, 1441. Compare: 428, 430, 432, 437, 439, 441, 444, 446, 622, 623. compare: 399, 414, 543. Compare Complex: 304, 305.CompareFormats: 807, 819, 820, 821, 822, 823, 824, 835, 836. CompareInt: 302, 303, 436, 512, 530, 538, 540, 553, 571, 605. CompareIntPairs: 436, 446, 501, 587, 611. CompareIntStr: 302, 303, 305.CompareNatFunc: 540, 611. CompareProc: 387, 388, 398, 400, 409, 410, 428, 429. CompareStr: 438, 439, 465, 470, 479, 611. CompareStringPtr: 424, 425, 611.Compare With: 528, 543. Complement Of: 506, 516.CompleteAdjectiveCluster: 876, 878, 1578, 1898, 1900. 1902. CompleteArgument: 1713, 1714, 1716, 1720, 1760. CompleteAtomicFormula: 1752, 1760. CompleteAttributeArguments: 876, 878, 1577, CompleteAttributes: 877, 878, 1735, 1758, 1895,

1900, 1902, 1903.

CompleteAttrIdentify: 867, 868.

CompleteClusterTerm: <u>876</u>, 878, 1579, 1898.

CompleteClusterType: 877, 878, 1898, 1900, 1902, 1903. CompleteFuncIdentify: 866, 868, 1463, 1906. CompleteMultiPredicativeFormula: 1747, 1756, 1757. CompletePragmas: 1373, 1377. CompletePredAntonymByAttr: 867, 868. CompletePredicativeFormula: 1751, 1756, 1757. CompletePredIdentify: 867, 868. CompletePredSynonymByAttr: 867, 868. Complete Right Side Of The Predicative Formula: 1746,1747, 1748. Complete Type: 876, 878, 1591, 1735, 1758, 1898, 1900, 1902, 1903. ComplexAdd: 286, 287. ComplexDiv: 295, 296, 299. ComplexInv: 298, 299.ComplexMult: 292, 293, 294. ComplexNeg: 290, 291, 293. ComplexNorm: 300, 301. ComplexSub: 288, 289. Cond: 147, 148. Conditional Definiens: 1126, 1135, 1248, 1366, 1429. ConditionalDefiniensObj: 1134, 1135. Conditional Definiers Ptr: 1134, 1248, 1310, 1366, 1565. ConditionalFormulaObj: 990, 991. ConditionalFormulaPtr: 990, 1296, 1658, 1668. Conditional Registration: 1167, 1172, 1273, 1323, 1371, 1452, 1461. Conditional Tail: 1746, 1764, 1770, 1772. COne: 277, 285, 299. ConjunctiveFormulaObj: 986, 987. ConjunctiveFormulaPtr: 986, 1296, 1658. Conjunctive Tail: 1766, 1767.Console Mode: 163.const: 7. ConstantDefinitionObj: 1077, 1078. ConstantDefinitionPtr: 1077, 1264, 1323, 1371, 1532. Construction Type: <u>1868</u>, 1870, 1871. constructor: 8, 606. ConstructorsProcessing: 92, 96, 98, 99, 100. continue: 773, 777, 784, 786, 1923. Contradiction: 1004. ContradictionFormulaObj: 1004, 1005. ContradictionFormulaPtr: 1004, 1295, 1650. coOverflow: 310, 490, 499.coOverFlow: 357, 510.

Copula: 1477.

copy: 200, 202, 229, 230, 694, 742, 774, 779, 782, 1366, 1367, 1368, 1371, 1377, 1388, 1443, 1471, 800, 801, 1375, 1376. 1591, 1616, 1643, 1646, 1665, 1667, 1668. Copy: 657, 660, 720, 721, 724, 725, 726, 793, count: 550.800, 1374. CountAll: <u>528</u>, 546, <u>579</u>, 596. CopyBinIntFunc: 579, 588. $create_supposition_block$: 1828, 1829. CopyCollection: 349, 365. $create_supposition_head$: 1828, 1829. CopyIntRel: 497, 502. CreateArgs: 1379, 1573, 1575, 1589, 1607, 1621, CopyInt2PairOfIntFunc: 597, 606. 1623, 1625, 1642, 1643, 1649. CopyItems: 322, 348.CreateBlock: 853, 859, 1385, 1403, 1825, 1828, CopyList: 322, 327, 349, 369, 387, 390. 1830, 1887, 1892, 1910, 1915. CopyNatFunc: $\underline{528}$, $\underline{529}$, $\underline{539}$. CreateExpression: 865, 866, 1531, 1773, 1776, CopyNatSet: 506, 513. 1898, 1900, 1902, 1903, 1911. CopyObject: 312, 315, 316, 329, 348, 539. CreateExpressionsVariableLevel: 1010, 1013, Copyright: 84, 85, 111. 1025, 1029. CopySequence: 552, 555. CreateItem: 853, 858, 1385, 1402, 1789, 1790, 1805, 1807, 1809, 1811, 1813, 1817, 1818, 1819, Corrected: 1597, 1598, 1604. 1821, 1825, 1828, 1829, 1858, 1869, 1870, 1871, Correctness: 1869, 1870, 1898, 1904, 1906. Correctness Conditions: 1565. 1881, 1883, 1885, 1887, 1891, 1892, 1894, 1898, 1904, 1906, 1908, 1910, 1911, 1913, 1914, 1915. CorrectnessConditionObj: 1160, 1161. CorrectnessConditionPtr: 1160, 1259, 1266, 1322, CreateSubexpression: 870, 872, 1669, 1671, 1714, 1720, 1735, 1760, 1776, 1898, 1900, 1902, 1903. 1323, 1371, 1430. CorrectnessConditionsObj: 1162, 1163. CtrlCPressed: 93, 117, 160, 164. CorrectnessConditionsPtr: 1162, 1267, 1322, CtrlSignal: 162, 163, 164, 165, 166. CurIndex: 452, 476. 1323, 1430. CurlyBracket: 882, 1191, 1501. CorrectnessConditionsSet: 1162, 1163, 1420. CorrectnessKind: 880, 882, 1160, 1161, 1162, CurPos: 128, 131, 134, 138, 151, 189, 191, 193, 846, 847, 864, 886, 889, 1038, 1197, 1259, 1275, 1260, 1281, 1286, 1424. CorrectnessKindLookupTable: 1281, 1282, 1286. 1320, 1322, 1323, 1324, 1325, 1326, 1371, 1383, 1387, 1391, 1393, 1394, 1395, 1396, 1398, 1399, CorrectnessName: 882, 1259, 1267, 1281, 1371. 1400, 1406, 1409, 1420, 1426, 1460, 1465, 1468, CorrectnessObj: 1158, 1159, 1160, 1162. 1469, 1473, 1474, 1476, 1479, 1490, 1491, 1493, CorrectnessPtr: 1158, 1267, 1371. 1498, 1501, 1502, 1505, 1507, 1512, 1513, 1515, coSortedListError: 310, 406, 408, 420, 472, 1519, 1521, 1522, 1527, 1529, 1532, 1533, 1537, 477, 573. 1541, 1542, 1545, 1548, 1549, 1553, 1556, 1558, Count: 322, 325, 326, 327, 329, 330, 331, 332, 333, 1559, 1563, 1572, 1575, 1577, 1579, 1580, 1581, 334, 335, 337, 338, 341, 342, 344, 345, 346, 348, 1583, 1588, 1589, 1590, 1609, 1614, 1615, 1617, 350, 352, 353, 354, 356, 357, 358, 362, 363, 365, 1618, 1620, 1623, 1625, 1626, 1628, 1630, 1632, 371, 373, 378, 379, 380, 381, 382, 383, 384, 386, 1633, 1634, 1635, 1636, 1637, 1638, 1639, 1640, 388, 389, 390, 391, 393, 395, 396, 400, 402, 403, 1641, 1648, 1650, 1651, 1652, 1653, 1656, 1659, 404, 407, 408, 412, 413, 415, 418, 419, 420, 421, 1660, 1661, 1662, 1663, 1666, 1679, 1725, 1748, 426, 432, 433, 448, 479, 486, 488, 489, 490, 491, 1757, 1802, 1825, 1828, 1829, 1830, 1887, 1892, 492, 493, 495, 499, 500, 501, 502, 507, 509, 510, 1895, 1898, 1908, 1910. 512, 513, 515, 516, 517, 521, 522, 524, 525, Current Year: 82, 85. 526, 529, 530, 537, 538, 540, 541, 542, 543, CurrPatternKind: 1880, 1881, 1883. 544, 546, 549, 551, 624, 626, 673, 740, 747, CurrPos: 130.758, 759, 760, 766, 767, 773, 779, 784, 789, 830, 831, 832, 833, 834, 839, 1015, 1018, 1020, CurrWord: 1921.1025, 1028, 1196, 1197, 1198, 1199, 1200, 1201, CurWord: 883, 886, 1381, 1395, 1397, 1398, 1413, 1205, 1206, 1215, 1216, 1217, 1223, 1227, 1229, 1422, 1424, 1426, 1463, 1473, 1491, 1493, 1498, 1233, 1234, 1238, 1240, 1248, 1251, 1254, 1256, 1501, 1502, 1503, 1507, 1513, 1517, 1542, 1543, 1257, 1261, 1262, 1263, 1265, 1271, 1272, 1274, 1544, 1545, 1546, 1548, 1549, 1551, 1552, 1555, 1335, 1336, 1339, 1340, 1341, 1342, 1345, 1348, 1556, 1557, 1558, 1559, 1560, 1561, 1572, 1575, 1349, 1354, 1356, 1357, 1359, 1361, 1364, 1365, 1588, 1609, 1620, 1622, 1623, 1624, 1626, 1628,

1632, 1634, 1639, 1640, 1641, 1648, 1653, 1677, DefinitionReferencePtr: 1051, 1251, 1314, 1356, 1680, 1683, 1688, 1689, 1691, 1692, 1695, 1696, 1556. 1702, 1706, 1713, 1714, 1716, 1718, 1720, 1722, DefinitionsProcessing: 92, 96, 98. 1725, 1727, 1729, 1731, 1733, 1739, 1744, 1746, DefPatternName: 1242, 1245, 1246, 1271, 1281. 1747, 1748, 1751, 1752, 1754, 1756, 1757, 1758, DefPos: 1887, 1892, 1910. 1760, 1762, 1766, 1767, 1768, 1769, 1770, 1774, defStandardMode: 1136, 1141, 1268, 1323, 1371. 1779, 1790, 1791, 1792, 1794, 1796, 1798, 1800, delete: 205, 639, 797, 799, 801, 1374. 1802, 1803, 1804, 1805, 1813, 1815, 1816, 1817, Delete: 38, 40, 349, 360, 361, 452, 463, 476, 630, 1818, 1819, 1821, 1822, 1823, 1827, 1828, 1830, 632, 635, 678, 718. 1832, 1838, 1842, 1843, 1845, 1847, 1853, 1855, $delete_prefix: 718.$ 1859, 1861, 1868, 1869, 1870, 1871, 1872, 1873, DeleteAll: 322, 326, 338, 346, 347, 370, 375, 1875, 1877, 1879, 1880, 1885, 1887, 1888, 1889, 389, 486, 495, 514, 529, 579, 586, 597, 604, 1891, 1892, 1893, 1894, 1895, 1897, 1898, 1900, 1471, 1591, 1616. 1904, 1906, 1908, 1910, 1915, 1917, 1919, 1920, DeleteElem: 506, 516, 519. 1921, 1922, 1923, 1924, 1925. DeleteExtItems: 370, 385. Curword: 1700, 1705, 1821, 1827, 1851. DeleteFile: 47, 48. CZero: 277, 296. DeleteInt: 568, 570.dArqList: 1606, 1607.DELPHI: 37, 54, 66, 87, 161, 166, 169. DARWIN: 87.Delta: 349, 350, 356, 357, 364, 365, 368, 369, 469, Data: 313.506, 507, 508, 510, 513, 514, 529, 530. Dct: 756, 757.Den: 256, 257, 259, 261, 263, 265, 267, 269, 273, .dct file: 756, 759, 760, 1189, 1190. 275, 277, 279, 281, 305. DctFile: 759. Depo: 1601, 1606, 1607. .dcx file: 756. DepoNbr: 1606, 1607. DEBUG: 198.Dereferencing: 320. debug_num: 198, 202, 203, 206, 208, 209, 210, destructor: 8. 212, 215, 219, 222, 224, 226, 230, 233, 235, dfEmpty: 1126, 1323, 1371, 1409, 1429, 1433. 237, 239, 244, 249, 251. dfEquals: 1126, 1323, 1371, 1479, 1480, 1484, DEBUGNUM: 198.1565. dec: 342, 352, 383, 398, 414, 421, 463, 479, 491, dfMeans: 1126, 1323, 1371, 1479, 1483, 1484, 1565. 534, 553, 561, 584, 593, 602, 706, 708, 776, dicthan: 714, 807. 799, 1025, 1029, 1185, 1326, 1393, 1401, 1586, DiffuseStatementObj: 1091, 1092. 1607, 1646, 1647, 1688, 1718, 1719, 1722, DiffuseStatementPtr: 1091, 1256, 1321, 1354, 1737, 1754, 1764, 1802. DefaultAllowed: 750, 805. $Dir: \underline{56}, \underline{58}, \underline{60}, \underline{62}.$ DefaultExt: 73, 74. DirectiveKind: 880, 882. defExpandableMode: 1136, 1139, 1268, 1323, 1371. DirectiveName: 882. DefExpressionObj: 1128, 1129. DirSeparator: 83, 628, 643, 644. DefExpressionPtr: 1128, 1130, 1131, 1132, 1133, disable_io_checking: 13, 109, 191. 1134, 1135, 1310, 1484, 1565. DisjunctiveFormulaObj: 988, 989. DEFINED: 12.DisjunctiveFormulaPtr: 988, 1296, 1658. DefiniensKindName: 1248, 1310. Disjunctive Tail: 1768, 1769.DefiniensObj: 1126, 1127, 1130, 1134. DisplayLine: 91, 109, 189, 191, 193, 846, 847, 864, 1259, 1324, 1325, 1326, 1371. DefiniensPtr: 1126, 1140, 1141, 1142, 1143, 1144, 1145, 1146, 1147, 1194, 1248, 1277, 1310, DisplayLineInCurPos: 109, 117. 1328, 1366, 1408. Dispose: 314, 339, 382, 386, 408, 420, 435, 456, DefiniensSort: 1126, 1127. 1174, 1176. Defining WayName: 1259, 1323. dispose: 683, 745, 755, 784, 836, 844, 845, 846, Definition: <u>1870</u>, 1888. 847, 891, 904, 906, 910, 912, 923, 925, 927, 929, DefinitionalBlock: 1887, 1923. 931, 933, 937, 939, 943, 945, 947, 949, 951, 959, DefinitionReference: 1047, 1052, 1251, 1314, 1356. 965, 971, 973, 975, 977, 979, 981, 983, 985, 999, DefinitionReferenceObj: 1051, 1052. 1025, 1029, 1039, 1041, 1045, 1058, 1065, 1067, $\{1926 \quad \text{Mizar Parser} \quad \text{INDEX} \quad 595$

```
1069, 1071, 1074, 1076, 1078, 1081, 1084, 1087,
                                                             dos: 37, 170.
     1092, 1094, 1097, 1098, 1107, 1111, 1116, 1118,
                                                             Down: 528, 534, 579, 593.
     1120, 1122, 1127, 1129, 1131, 1133, 1135, 1137,
                                                             DrawArticleName: 110, 114, 194.
     1139, 1141, 1143, 1145, 1147, 1149, 1153, 1155,
                                                             DrawErrorsMSg: 126.
     1157, 1159, 1168, 1170, 1172, 1174, 1176, 1178,
                                                             DrawErrorsMsg: 125, 197.
     1182, 1184, 1192, 1276, 1327, 1372, 1443.
                                                             DrawIOResult: 70, 122, 123.
                                                             DrawMessage: 71, 72, 76, 118, 119, 121, 123,
DisposeFormats: 840.
                                                                  131, 136, 155, 156, 642.
DisposePrf: 883, 888, 893.
                                                             DrawMizarScreen: 110, <u>111</u>, 193.
DisposeStr: 440, 456, 462, 463, 484, 611, 621.
Dispose WSLookup Tables: 1282, 1327.
                                                             DrawPass: 115, <u>116</u>.
                                                             DrawStr: 110, 111.
Div: 225, 226, 233, 251.
                                                             DrawTime: 115, 116, 189.
div: 9.
                                                             DrawTPass: 111.
diva: 259.
                                                             DrawVerifierExit: 115, 116, 191.
DivA: 237, 250, 251.
                                                             DT: 648, 659.
Divides: 254, 255.
                                                             dupAccept: 452, 465.
div1: 224, 227.
                                                             dupError: 452, 454, 459.
Div1: \underline{224}.
                                                             dupIqnore: 452, 459.
Do-support: 1747.
                                                             Duplicates: 428, 429, 432, 433, 434, 435, 506,
do: 1747.
                                                                  507, 510, 512.
does: 1747.
                                                             DWORD: 163, 165, 166.
dol_Allowed: 1407, 1409, 1528, 1529, 1534, 1634.
                                                             Dynamic Array: 322.
Done: 28, 312, <u>314</u>, 322, 328, 339, 346, 349, 351,
                                                             e: 1281.
     370, 372, 382, 386, 408, 409, 411, 420, 435, 449,
                                                             EBNF: 1035.
     450, 452, 455, 456, 486, 489, 506, 509, 536, 552.
                                                             ec: 692.
     557, <u>579</u>, 582, <u>597</u>, 600, <u>619</u>, 621, <u>653</u>, 656, <u>661</u>,
                                                             EClusterObj: 1169, 1170.
     663, 680, 683, 684, 687, 693, 696, 697, 722, 730,
                                                             EClusterPtr: 1169, 1273, 1323, 1371, 1452.
     731, 732, 733, 735, 745, 755, 760, 764, 769, 784,
                                                             EE: 648, 659, 665, 666, 668, 669.
     792, 806, 836, 839, 840, 844, 845, 846, 847, 853,
                                                             eEnd: 648, 665, 667, 668, 670, 836, 1287, 1289,
     856, 863, 864, 873, 875, 888, 891, 903, 904, 905,
                                                                  1297, 1302, 1304, 1310, 1314, 1323, 1326.
     906, 907, 908, 909, 910, 911, 912, 922, 923, 924,
                                                             EI: 648, 659, 662.
     925, <u>926</u>, 927, <u>928</u>, 929, <u>930</u>, 931, <u>932</u>, 933, <u>936</u>,
                                                             elAdjective: 613, 1199.
     937, 938, 939, 942, 943, 944, 945, 946, 947,
                                                             elAdjectiveCluster: 613, 1200.
     948, 949, 950, 951, 958, 959, 960, 961, 962,
                                                             elAncestors: 613, 1271.
     963, 964, 965, 970, 971, 972, 973, 974, 975,
                                                             ElapsedTime: 174, <u>175</u>, 180.
     976, 977, 978, 979, 980, 981, 982, 983, 984,
                                                             elArguments: 613, 1215, 1216, 1227.
     985, 998, 999, 1010, 1012, 1025, 1029, 1036,
                                                             elArticleID: 613.
     1037, 1039, \underline{1040}, 1041, \underline{1044}, 1045, \underline{1057}, 1058,
                                                             elBlock: 613, 1197, 1326.
     <u>1059</u>, 1060, <u>1061</u>, 1062, <u>1064</u>, 1065, <u>1066</u>, 1067,
                                                             elConditions: 613, 1262, 1265, 1272.
     <u>1068</u>, 1069, <u>1070</u>, 1071, <u>1073</u>, 1074, <u>1075</u>, 1076,
                                                             elCorrectnessConditions: 613, 1267.
     1077, 1078, 1080, 1081, 1083, 1084, 1086, 1087,
                                                             elDefiniens: 613, 1248, 1310.
     1089, 1090, 1091, 1092, 1093, 1094, 1095, 1096,
                                                             elDirective: 613.
     1097, 1098, <u>1106</u>, 1107, <u>1110</u>, 1111, <u>1115</u>, 1116,
                                                             ElemLookup Table \colon \quad \underline{1281}, \ 1282, \ 1283.
     1117, 1118, 1119, 1120, 1121, 1122, 1126, 1127,
                                                             ElemNr: 506, 527.
     <u>1128</u>, 1129, <u>1130</u>, 1131, <u>1132</u>, 1133, <u>1134</u>, 1135,
                                                             elEnviron: 613.
     1136, 1137, 1138, 1139, 1140, 1141, 1142, 1143,
                                                             elEquality: 613, 1263, 1323.
     1144, 1145, 1146, 1147, 1148, 1149, 1152, 1153,
                                                             elFieldSegment: 613, 1271, 1323.
     <u>1154</u>, 1155, <u>1156</u>, 1157, <u>1158</u>, 1159, <u>1160</u>, 1161,
     <u>1162</u>, 1163, <u>1164</u>, 1165, <u>1167</u>, 1168, <u>1169</u>, 1170,
                                                             elFormat: 613, 836, 838.
     <u>1171</u>, 1172, <u>1173</u>, 1174, <u>1175</u>, 1176, <u>1177</u>, 1178,
                                                             elFormats: 613, 836, 839.
     <u>1179</u>, 1180, <u>1181</u>, 1182, <u>1183</u>, 1184, 1192, <u>1194</u>,
                                                             elIdent: 613.
     1195, 1276, <u>1277</u>, 1278, 1282, 1327, <u>1328</u>, 1329,
                                                             elItem: 613, 1275, 1324, 1325.
     1372, 1443, 1471, 1616, 1665, 1667.
                                                             elIterativeStep: 613, 1256, 1321.
```

elLabel: 613, 1247, 1311. entry_is_uninitialized: 1842, 1855, 1863. elLink: 613, 1252, 1318. $entry_is_unitialized$: 1842. EnvFileExam: 72. elLoci: 613, 1240, 1323. elLociEquality: 613, 1274, 1323. EnvFileName: 36, 72, 75, 78, 643, 893, 1388. elLocus: 613, 1239.Environment variable: 65, 617. elNegatedAdjective: 613, 1199. EnvMizFiles: 617, 645. elPartialDefiniens: 613, 1248, 1310. eof: 625, 658, 742, 743, 803. EOT: 763, 774, 789, 884, 1678, 1802, 1817, 1827, elPriority: 613, 836, 839. elProposition: 613, 1249, 1323. 1888, 1893, 1910, 1921, 1922, 1923. EOTX: 648, 658, 662, 665, 666, 668, 669. elProvisionalFormulas: 613, 1261, 1323. elRedefine: 613, 1268, 1269, 1270, 1323. EQ: 648, 659, 666, 669. elRightCircumflexSymbol: 613, 1228, 1244. EqualitiesProcessing: 92, 96, 98. elSchematic Variables: 613, 1261. EqualitySym: 882, 1191, 1855. elScheme: 613, 1261. Equals: 511, 521, 522, 537, 538. else: 10, 12. Equating: 1080, 1263, 1323, 1371, 1537. ELSE: 12, 83, 86, 157. *Erase*: 697. $else_def\colon \ \underline{1}2.$ erase: 142. $else_if_def: 12.$ EraseErrors: 141, $\underline{142}$. ELSEIF: 12.EraseFile: 47, 48, 50, 626, 693, 697. Err: 648, 659. elSelector: 613, 1271, 1323. elSetMember: 613.errBadXMLToken: 647, 664. elSkippedProof: 613. errElMissing: 647, 670. elSubstitution: 613. errElRedundant: 647, 670. elSymbol: 613, 747, 760, 1192. errhan: 127, 151, 154, 185, 646, 674, 748, 807, elSymbolCount: 613, 746. 841, 879, 895, 1030, 1379, 1672. elSymbols: 613, 747, 760. ErrImm: 130, 131, 155, 886, 1398, 1413, 1417, elTypeList: 613, 1238. 1422, 1424, 1487, 1556, 1557, 1579, 1633, elTypeSpecification: 613, 1261, 1268, 1270, 1323. 1634, 1638, 1639, 1677, 1684, 1794, 1798, elUnknown: 613, 1283. 1828, 1881, 1883, 1885, 1898, 1900, 1902, elVariable: 613, 1202, 1293, 1315, 1323. 1904, 1917, 1921, 1925. elVariables: 613, 1205, 1257, 1261. errMissingXMLAttribute: 647, 691. elVocabularies: 613, 745. ErrMsg: 123, 124, 155. ErrNr: 128, 130, 131, 132, 133, 193. elVocabulary: 613, 745.EmptyNatFunc: 611. Error: 130, 131, 680, 681, 684, 685, 1433, 1456, 1543, 1575, 1589, 1599, 1600, 1603, 1604, 1605, EmptyParameterList: 112, 113, 193. 1625, 1627, 1629, 1642, 1643, 1677, 1680, 1889. EN: 648.enable_io_checking: 13, 191. Error, 58: 1828. Error, 65: 1638. $ENABLE_PROCESSED_INPUT$: 163. end: 10, 11, 12. Error, 73: 1543. Error, 146: 1556, 1557. $end_empty_tag: \underline{669}.$ Error, 151: end_if: 12. 1589. Error, 152: end_mdebug: 11. 1623. Error, 153: $end_start_tag: \underline{669}.$ 1642.endcases: 10. Error, 157: 1639. Error, 175: ENDIF: 11, 12, 83, 86, 154, 157, 160, 161, 165, 1574. 166, 169, 170, 198, 313. Error, 176: 1625.Error, 178: endif: 12. 1456.EndOfIdent: 772, 773, 776, 777, 782, 784, 786. Error, 181: 1634. EndOfPhrase: 778, 779, 780. Error, 182: 1627. Error, 183: EndOfSymbol: 772, 773, 779, 780. 1729.EndOfText: 764, 788, 789, 792, 796, 803. Error, 184: 1629, 1731. EnlargeBy: 506, 515, 528, 537, 538, 539. Error, 185: 1589.

 $\{1926 \quad \text{Mizar Parser} \quad \text{INDEX} \quad 597$

```
Error, 186:
             1877.
                                                       Error, 384:
                                                                    1703, 1794, 1873, 1877, 1915.
Error, 200:
                                                       Error, 385:
                                                                     1856, 1900.
             777.
Error, 201:
             777.
                                                       Error, 386:
                                                                     1810.
Error, 202:
                                                       Error, 387:
             785.
                                                                     1742.
Error, 203:
                                                       Error, 388:
             787.
                                                                     1813, 1814.
Error, 213:
             1922.
                                                       Error, 389:
                                                                     1830, 1920.
Error, 214:
                                                       Error, 390:
             1680.
                                                                     1906.
Error, 215:
                                                       Error, 391:
             1680.
                                                                     1885.
Error, 223:
                                                       Error, 392:
             1900, 1902.
                                                                     1885.
Error, 231:
             1823.
                                                       Error, 395:
                                                                     1800.
Error, 232:
             1825, 1829.
                                                       Error, 396:
                                                                     1760.
Error, 251:
                                                       Error, 397:
             1633.
                                                                     1716, 1721.
Error, 256:
             1709, 1908.
                                                       Error, 398:
                                                                     1727.
Error, 273:
             1889.
                                                       Error, 399:
                                                                     1847.
Error, 275:
             1711.
                                                       Error, 400:
                                                                     1908.
             1693, 1783, 1808, 1810, 1812, 1813,
Error, 300:
                                                       Error, 401:
                                                                     1747, 1749, 1757, 1769.
    1814, 1835, 1873, 1877, 1906, 1911, 1917.
                                                       Error, 402:
                                                                     1748, 1757, 1767.
Error, 301: 1853.
                                                       Error, 403:
                                                                     1785.
Error, 302:
            1722, 1754, 1849, 1904.
                                                       Error, 404:
                                                                     1904.
Error, 303:
             1838.
                                                       Error, 406:
                                                                     1898.
Error, 304:
                                                       ErrorAddr:
             1861.
                                                                     155.
Error, 305:
             1866.
                                                       ErrorCode: 145, 146, 147, 148, 149, 150, 155.
Error, 306:
             1725, 1845, 1895.
                                                       ErrorNbr: 128, 131, 134, 137, 155, 189, 191, 193,
Error, 307:
             1794, 1798.
                                                            197, 846, 847, 864, 1259, 1371.
Error, 308:
             1791, 1792, 1796.
                                                       ErrorRecovery: 661, 662, 664, 665, 666, 668, 669.
Error, 309:
             1758.
                                                       ErrorReport: 128.
Error, 310:
             1700, 1705, 1851.
                                                       Errors: 132, 133, 135, 140, 141, 142, 144.
Error, 312:
             1794.
                                                       ErrorSymbol: 763, 777, 786, 787.
Error, 313:
             1798.
                                                       errWrongXMLElement: 647, 651.
                                                       eStart: 648, 665, 669, 670, 1192, 1293, 1301,
Error, 314:
             1881, 1883.
Error, 315:
             1840.
                                                            1307, 1310, 1311, 1315, 1320, 1321, 1323,
Error, 320:
             1706.
                                                            1324, 1325, 1326.
Error, 321:
             1752.
                                                       ET: 648, 659, 665, 666, 667, 668, 669.
Error, 330:
             1680.
                                                       ex: 234, 235, 239, 240, 241, 242, 244, 245, 246,
Error, 336:
             1770.
                                                            247, 249, 251.
Error, 340:
             1744.
                                                       ExactlyTermObj: 946, 948, 949.
Error, 350:
             1785, 1787.
                                                       ExactlyTermPtr: 948, 1029, 1236, 1301, 1348,
Error, 351:
                                                            1583.
             1823.
Error, 360:
                                                       exAdjectiveCluster: 28, 869, 1895, 1898, 1900,
             1808.
Error, 361:
                                                            1902.
             1810.
Error, 362:
             1917.
                                                       ExampleObj: 1083, 1084.
                                                       ExamplePtr: 1083, 1266, 1323, 1371, 1541, 1542.
Error, 363:
             1698, 1864.
Error, 364:
                                                       Exchange: 452, 464, <u>1597</u>, 1598, 1603, 1604.
             1917.
Error, 370:
             1696, 1714, 1764, 1796, 1808, 1843,
                                                       ExchangeItems: 453, 464, 479.
    1845, 1859, 1917.
                                                       exclude: 1424.
Error, 371: 1762, 1810, 1917.
                                                       Exercises: 1031.
Error, 372:
             1703, 1915.
                                                       exFormula: 869, 1248, 1310, 1366, 1484, 1565,
Error, 373:
             1698, 1864.
                                                            1773, 1776.
Error, 380:
             1807, 1808, 1811, 1812, 1906.
                                                       Existential Assumption: 1150, 1789, 1817, 1891.
Error, 381:
             1875, 1879.
                                                       Existential Assumption Obj: 1156, 1157.
Error, 382:
             1881, 1883, 1902, 1911.
                                                       ExistentialAssumptionPtr: 1156, 1265, 1323,
                                                            1371, 1437.
Error, 383:
             1845.
```

ExistentialFormula: 1742, 1760. 1586, 1587, 1588, 1589, 1590, 1591, 1593, 1597, 1609, 1610, 1611, 1612, 1613, 1614, 1615, 1616, ExistentialFormulaObj: 1002, 1003.1617, 1618, 1619, 1620, 1621, 1622, 1623, 1624, ExistentialFormulaPtr: 1002, 1295, 1667. ExistentialRegistration: 1167, 1170, 1273, 1323, 1625, 1626, 1627, 1628, 1629, 1630, 1631, 1632, 1633, 1634, 1635, 1636, 1637, 1638, 1639, 1640, 1371, 1409, 1452. ExitProc: 152, 153, 155, 157, 158, 159, 191, 195. 1641, 1642, 1643, 1644, 1645, 1646, 1647, 1648, exNull: 869, 1895, 1897, 1898. 1649, 1650, 1651, 1652, 1653, 1654, 1655, 1656, 1657, 1658, 1659, 1660, 1661, 1662, 1663, 1664, Expandable Mode Definition Obj: 1138, 1139.1665, 1666, 1667, 1668. ExpandableModeDefinitionPtr: 1138, 1268, 1323, extSubexpPtr: 1568, 1671.1371, 1444. exType: 869, 1773, 1898, 1900, 1902, 1903. ExpansionsProcessing: 92, 96, 98. fAnother: 322, 346, 347, 349, 364, 506, 515, ExpKind: 865, 866, 869, 870, 871, 1010, 1011, 516, 517, 528, 538. 1128, 1129, 1248, 1310, 1531, 1669, 1670, fArgsNbr: 817, 818, 823, 838, 1690, 1691, 1737, 1895, 1898, 1904. Explicitly Qualified Segment Obj: 905, 906.fAt: 693, 709, 710, 713. ExplicitlyQualifiedSegmentPtr: 905, 1018, 1205, fBlockKind: 853, 854, 859, 1385, 1386, 1403. 1293, 1339, 1348, 1471, 1616, 1665. fBuffCount: 680, 682, 696, 699. ExpName: 1248, 1310. Expression Obj: 870, 871, 872, 1669. fBuffInd: 680, 682, 696, 698, 699. fCapacity: 452, 454, 469, 473, 480, 552, 554, 555, ExpressionPtr: 850, 865, 870. 556, 558, 565, 567, 569, 579, 581, 583, 585, exResType: 869, 1911. 588, 597, 599, 601, 603, 606. Ext: 56, 58, 60, 62. fCh: 1683. extBlockObj: 1385, 1386, 1393, 1394, 1395, 1396, FClusterObj: 1173, 1174.1397, 1398, 1399, 1401, 1402, 1403. FClusterPtr: 1173, 1273, 1323, 1371, 1452. extBlockObj.ProcessBegin: 1394.fCompare: 387, 388, 389, 390, 391, 392, 400, extBlockPtr: 1383, 1385, 1391, 1403, 1456, 1542, 401, 402, 405, 406, 407, 409, 410, 412, 413, 1559, 1560, 1561. 425, 428, 429, 430. ExtBlockPtr: 1410, 1424, 1475. fCount: 452, 454, 455, 458, 461, 462, 463, 465, exTerm: 869, 1248, 1310, 1366, 1484, 1565, 470, 473, 474, 480, 482, 552, 554, 556, 557, 558, 1773, 1897, 1898. 559, 560, 561, 563, 564, 565, 566, 567, 569, 571, extExpressionObj: 1669, 1670, 1671. 575, 576, 578, 579, 581, 582, 583, 584, 585, 586, extExpressionPtr: 1531, 1669. 587, 588, 595, 596, 597, 599, 600, 601, 602, extItemObj: 1405, 1406, 1429, 1457, 1459, 1460, 603, 604, 605, 606, 744, 745, 839. 1461, 1462, 1463, 1465, 1466, 1468, 1469, fCurrentLine: 792, 793, 795, 796, 797, 798, 799, 1470, 1471, 1473, 1474, 1475, 1476, 1479, 1480, 800, 801, 803, 805, 806. 1481, 1483, 1484, 1485, 1486, 1487, 1490, 1491, fDelta: 349, 366. 1492, 1493, 1494, 1495, 1498, 1499, 1501, 1502, fDuplicate: 452, 454, 459, 465. 1503, 1504, 1505, 1506, 1507, 1509, 1510, 1512, 1513, 1514, 1515, 1517, 1519, 1521, 1522, 1523, fEl: 693, 703, 705, 707, 708, 711. fElem: 506, 519, 520, 528, 535, 547, 550. 1524, 1525, 1527, 1529, 1530, 1531, 1532, 1533, 1534, 1535, 1536, 1537, 1538, 1539, 1540, 1541, fErrNbr: 91, 108, 109, 117. fErrNr: 1672, 1677, 1679, 1683, 1684. 1542, 1543, 1544, 1545, 1546, 1548, 1549, 1551, fExpKind: 865, 866, 870, 871, 1531, 1669, 1670. 1552, 1553, 1555, 1556, 1557, 1558, 1559, 1560, 1561, 1563, 1564, 1565. fExt: 627, 628. extItemPtr: 1402, 1405.fExtCount: 370, 371, 374, 378, 379, 380, 383, 384, Extract: 722, 724, 728, 737. 385, 386, 418, 419, 420, 421, 422. ExtractFileDir: 59, 60, 643. FF: 1842, 1861, 1863. ExtractFileExt: 54, 61, 62, 73, 74, 135, 140, 741. fFileBuff: 680, 682, 683, 696, 698, 699. $ExtractFileName: 54, 61, \underline{62}, 75.$ fFirstChar: 753, 758, 779. ExtractFilePath: 54. fFunc: 506, 513, 514, 521, 522, 523, 524, 525, extSubexpObj: <u>1568</u>, 1570, 1571, 1572, 1575, 1576, 526, <u>528</u>, <u>529</u>, 537. 1577, 1578, 1579, 1580, 1581, 1582, 1583, 1585, fIdent: 619, 620.

fIdents: 764, 766, 768, 769, 784, 893. FinishChoiceTerm: 876, 878, 1631, 1706, 1709, 1711. FieldSegmentObj: 1106, 1107.FieldSegmentPtr: 1106, 1271, 1323, 1371, 1514. FinishClusterTerm: 866, 868, 1462, 1900. FinishClusterType: 867, 868, 1898, 1900, 1902, FieldSymbolObj: 1104, 1105.FieldSymbolPtr: 1104, 1271, 1323, 1371, 1513. 1903. FinishCompactStatement: 866, 868, 1561, 1832. FIL: 625, 626. file: $\underline{70}$. FinishCondition: 866, 868, 1539, 1785. FinishConsequent: 866, 868, 1461, 1898, 1900, File: 680. 625. 1902, 1903. File, .fil: FinishConstructionType: 866, 868, 1486, 1868. File, .dct: 1189, 1190. FinishDefaultTerm: 866, 868, 1538, 1813. File, .frt: 1030. FinishDefiniens: 866, 868, 1565, 1873, 1877. File, .idx: 893, 1192. FinishDefinition: 853, 857, 1870. File, .ini: 639. FinishDrawing: 110, 111, 197. File, .prf: 887. File, .wsx: FinishEquality: 877, 878. 1030. FinishExemplifyingTerm: <u>866</u>, 868, 1542, 1818. FileAge: 51, 52. FinishExemplifying Variable: 866, 868, 1541, 1818. FileDescr: 619, 620, 621. FinishExistential: 876, 878, 1667, 1742. FileDescrCollection: 618, 622, 623, 624, 625, 626. FinishFields: 866, 868, 1510, 1864. FileExam: 70, 195, 630, 632, 635, 640, 743, 1388. FinishFixedSegment: 866, 868, 1471, 1783. FileExists: 47, 48. FinishFixedVariables: 866, 868, 1473, 1782. FileMode: 70.FinishFlexConjunction: 876, 878, 1660, 1767. FileName: 122, 123, 134, 135, 136, 139, 140, 634, 635, 636, 638. FinishFlexDisjunction: 876, 878, 1659, 1769. FinishForgetfulTerm: 876, 878, 1629, 1709. FillChar: 313, 732, 734. Find: 387, 392, 401, 402, 403, 404, 405, 409, 412, FinishFraenkelTerm: <u>876</u>, 878, 1617, 1703. 413, 414, 426, 452, 458, 465, 470, 568, 569, 570, FinishFuncReduction: 867, 868, 1904. 571, 572, 574, 826, 827, 828, 829, 830. FinishFunctorPattern: <u>866</u>, 868, 1503, 1847, fIndex: 387, 388, 389, 390, 391, 393, 394, 396, 397, 1849, 1851. 400, 402, 404, 405, 408, 409, 412, 413, 414, 415, FinishGuard: 866, 868, 1484, 1875, 1879. 419, 420, 421, 426, 758, 778, 826, 827, 828, 829. FinishHypothesis: 866, 868, 1540, 1778, 1781. FindInterval: 409, 414. FinishingPass: 93, 117, 189. FindRight: 409, 413, 421. FinishIterativeStep: 866, 868, 1564, 1834. fInFile: 807, 837. FinishJustification: 867, 868, 1803, 1832, 1833, fInfinitive: 28, 722, 723. 1834. FinishAdjectiveCluster: 877, 878, 1724. FinishLocusType: 866, 868, 1530, 1804. FinishAggregateTerm: 876, 878, 1625, 1698. FinishLongTerm: 876, 878, 1597, 1714. FinishAggrPattSegment: <u>866</u>, 868, 1514, 1866. FinishModePattern: <u>866</u>, 868, 1494, 1838. FinishAntecedent: 866, 868, 1461, 1902. FinishMultiPredicativeFormula: 876, 878, 1645, FinishArgList: 876, 878, 1610, 1714. 1747.FinishArgument: 876, 878, 1585, 1607, 1639, 1705, FinishOtherwise: 866, 868, 1480, 1873, 1875, 1713, 1714, 1716, 1721, 1737, 1738. 1877, 1879. FinishAssumption: 867, 868, 1475, 1779, 1787. FinishPass: 188, <u>189</u>. FinishAtSignProof: 853, 857, 1802. FinishPostqualifyingSegment: 876, 878, 1692. FinishAttributeArguments: 876, 878, 1577, 1897. FinishPostQualifyingSegment: 1616. FinishAttributePattern: 866, 868, 1491, 1845. FinishPredicatePattern: 866, 868, 1499, 1853. FinishAttributes: 877, 878. FinishPredicativeFormula: <u>876</u>, 878, 1642, 1751. FinishAttributiveFormula: 876, 878, 1647, 1758. FinishPrefix: 866, 868, 1506, 1859. FinishBinaryFormula: 876, 878, 1658, 1766, FinishPrivateConstant: <u>866</u>, 868, 1532, 1811. 1768, 1770. FinishPrivateFormula: 876, 878, 1649, 1762. FinishBracketedTerm: 876, 878, 1623, 1690, 1705. FinishPrivateFuncDefinienition: 866, 868, 1535, FinishChoice: 867, 868, 1785. 1807.

FinishPrivatePredDefinienition: 866, 868, 1536, FlexaryDisjunctiveFormulaPtr: 996, 1295, 1659. 1809. FlexConjunctiveTail: 1767, 1768.FinishPrivateTerm: 876, 878, 1621, 1696. FlexDisjunctive Tail: 1769, 1770. FinishQualifiedSegment: <u>876</u>, 878, 1665, 1739. fLexem: 751, 752, 759, 760, 762, 766, 767, 768, FinishQualifiedTerm: 876, 878, 1582, 1689. 773, 784, 789, 790, 886, 889. FinishQualifyingFormula: 876, 878, 1646, 1758. fLine: 91, 108, 109, 117, 792, 802. FinishQuantified: 877, 878, 1742, 1744. fList: 452, 454, 456, 461, 462, 463, 464, 465, 467, FinishReconsideredTerm: 866, 868, 1537, 1813.468, 470, 471, 473, 477, 478, 479, 480, 552, 554, FinishReconsidering: 867, 868, 1813. 556, 558, 559, 561, 563, 564, 565, 566, 567, 569, 571, 575, 576, 578, 579, 581, 583, 584, 585, 587, FinishReferences: <u>867</u>, 868, 1791. FinishReservation: 867, 868, 1911. 588, 592, 593, 594, 595, 596, 597, 599, 601, 602, FinishReservationSegment: 866, 868, 1527, 1911. 603, 605, 606, 610, 629, 630, 631, 632, 633, 634, 635, 636, 637, 638, 744, 745, 839, 1729, FinishRestriction: 876, 878, 1657, 1744. FinishRightSideOfPredicativeFormula: 876, 878, 1731, 1746, 1751, 1842, 1855, 1863. 1643, 1746. fName: 110, 114, 622, 625, 626, 627, 628, 629, 630, 631, 632, 633, 717, 741, 742, 825, 836, 839. FinishSample: 877, 878, 1703. fNatFunc: 528, 541, 542, 543. FinishScanning: 883, 893. FinishSchemeDeclaration: 867, 868, 1915. fObject: 451, 456, 461, 468, 471, 473, 478, 744, 745. FinishSchemeDemonstration: 853, 860, 1385, Foo: 306. 1401, 1920. foo: 306. FinishSchemeHeading: 867, 868, 1915. ForgetfulFunctor: 884, 1880. FinishSchemePremise: 866, 868, 1525, 1919. ForgetfulFunctorTermObj: 938, 939. FinishScheme Qualification: 866, 868, 1519, 1917.ForgetfulFunctorTermPtr: 938, 1029, 1231, 1301, FinishSchemeReference: 867, 868, 1796. 1348, 1629. FinishSchemeSegment: <u>866</u>, 868, 1523, 1917. fOrigin: 322, 348.FinishSchemeThesis: 866, 868, 1524, 1915. FormatDateTime: 82. FinishSchLibraryReferences: 867, 868, 1798. formats: 807, 1379. FinishSelectorTerm: 876, 878, 1627, 1708. FormatsProcessing: 92, 96, 98, 99, 100. FinishSentence: 867, 868, 1775, 1776. FormulaExpression: <u>1773</u>, 1775, 1809, 1873, FinishSethoodProperties: 866, 868, 1492, 1908. 1875, 1879, 1915. FinishSimpleFraenkelTerm: 876, 878, 1619, 1711. FormulaExpressionObj: 968, 970, 974, 976, 978, FinishSimpleJustification: 866, 868, 1560, 1800. 980, 982, 984, 998, 1004, 1006, 1008. FinishSpecification: 866, 868, 1485, 1856. FormulaKindLookupTable: 1281, 1282, 1285. FinishTerm: 876, 878, 1586, 1720. FormulaName: 1206, 1207, 1208, 1209, 1210, FinishTheLibraryReferences: 867, 868, 1794. 1211, 1212, 1213, 1214, 1215, 1216, 1217, 1218, FinishTheorem: 867, 868, 1790, 1913, 1914. 1219, 1220, 1221, 1281. FinishTheoremBody: 867, 868, 1913, 1914. FormulaPtr: 944, 945, 968, 982, 983, 984, 985, FinishType: 876, 878, 1589, 1727. 986, 987, 988, 989, 990, 991, 992, 993, 994, FinishUniversal: 876, 878, 1668, 1744. 995, 996, 997, 998, 999, 1000, 1001, 1002, Finish Visible: 867, 868, 1837, 1843. 1003, 1010, 1025, 1027, 1044, 1045, 1068, 1069, 1075, 1076, 1132, 1133, 1194, 1207, Finitary metatheory: 1100, 1102. 1248, 1277, 1295, 1328, 1345, 1366, 1385, 1392, fInt: 441, 442. 1428, 1440, 1568, 1643. FirstThat: 441, 448. fItemKind: 853, 858, 863, 864, 1385, 1402. FormulaSort: 968, 1281, 1285. Fixed Variables: 1782, 1785, 1787, 1894, 1910. FormulaSubexpression: 1691, 1703, 1742, 1744, fKey: 441, 445, 446, 448. 1772, 1773. forward: 1689, 1690, 1691, 1746, 1802, 1805. fKind: 722, 723, 1405, 1406. FoundToken: 772, 773, 778, 779, 780. fLatAtom: 528, 539. fOutFile: 811, 838. fLeftArgsNbr: 815, 816, 818, 824, 838. FlexaryConjunctiveFormulaObj: 994, 995. fParam: 633, 634, 636, 637, 638. FlexaryConjunctiveFormulaPtr: 994, 1295, 1660. fParenthCnt: 1688, 1713, 1714.FlexaryDisjunctiveFormulaObj:996, <u>997</u>. fPassName: 111.

FPC: 37, 54, 66, 87, 154, 160, 161, 165, 170. FunctorialRegistration: 1167, 1174, 1273, 1323, 1371, 1452, 1462. fPhrase: 764, 768, 769, 773, 774, 776, 777, 779, FunctorName: 1187, 1189, 1190, 1227, 1243, 782, 786, 787, 793, 800. fPhrasePos: 764, 768, 773, 777, 782, 787, 789, 1348, 1365. 793, 795, 796, 800, 801, 803. FunctorPatternName: 1243, 1244, 1307, 1309. FunctorPatternObj: 1121, 1122.fPos: 693, 711, 712, 761, 762, 768, 789, 790, 796, FunctorPatternPtr: 1121, 1146, 1147, 1241, 1277, 800, 886, 889, 1677, 1680. 1307, 1323, 1365, 1445. fPriority: 722, 723. FunctorSegment: 1064, 1067, 1261, 1323, 1371. fpSignal: 160.FunctorSegmentObj: 1066, 1067.Fraenkel term: 944, 1233. FunctorSegmentPtr: 1066, 1261, 1323, 1371, 1523. FraenkelTermObj: 944, 945. FunctorSort: 1121, 1122. FraenkelTermPtr: 944, 1029, 1233, 1301, 1348, fVal: 693, 709, 710, 713. fVocFile: 717, 742.Free: 312, 314, 349, 355, 361. fW: 1683.FreeAll: 322, 328, 340, 351, 370, 376, 671. qAddSymbolsSet: 1676. FREEBSD: 87. *gAntecedent*: 1451, 1452, 1461. FreeExtItems: 370, 372, 386. gAttrColl: <u>1458</u>, 1459, 1461, 1578. FreeItem: 322, 339, 341, 355, 361, 437, 440. gBlockPtr: 847, 850, 854, 856, 859, 1383, 1403, FreeItemsFrom: 322, 340, 341, 370, 382, 387, 1410, 1424, 1456, 1475, 1542, 1559, 1560, 408, 409, 420. 1561, 1789, 1790, 1802, 1805, 1807, 1809, FreeMem: 343, 379, 397, 480, 484, 494, 538, 567, 1811, 1813, 1815, 1817, 1818, 1819, 1820, 1821, 585, 656, 687, 806. 1825, 1828, 1829, 1830, 1858, 1869, 1870, 1871, fRepr: 722, 723.1881, 1883, 1885, 1887, 1888, 1891, 1892, 1894, fRightArgsNbr: 813, 814, 816, 818, 822, 824, 838. 1898, 1904, 1906, 1908, 1910, 1911, 1913, 1914, fRightSymbolNr: 817, 818, 823, 838. 1915, 1920, 1922, 1924. .frt file: 1030. GCD: 27, 234, 235, 237, 253, 255. fSing: 349, 366. gcd: 259.fSorted: 452, 454, 458, 470, 472, 477, 481, 482, 743. gClusterSort: <u>1408</u>, 1409, 1452, 1461, 1462. fSourceBuff: 792, 805, 806. gClusterTerm: <u>1451</u>, 1452, 1462. fSourceBuffSize: 792, 805, 806. gComment: 109, 117.fSourceFile: 792, 796, 803, 805, 806. *qConsequent*: 1451, 1452, 1461. fStart: 792, 802. gConstructorNr: 1448, 1450, 1491, 1493, 1494, fStr: 317, 318, 424, 752, 758, 759, 760, 762, 766, 1498, 1499, 1501, 1502, 1503, 1507, 1509, 1510. 767, 773, 779, 780, 789, 790, 886, 889. gCorrCondSort: 1424, 1430. fString: 451, 456, 461, 462, 463, 465, 467, 470, gCorrectnessConditions: <u>1420</u>, 1424, 1429, 1430, 473, 477, 479, 744, 745. 1444, 1452, 1455, 1543, 1544, 1565. fSymbol: 811, 812, 814, 816, 818, 819, 820, gDefiniens: 1408, 1409, 1429, 1444, 1445, 1446, 821, 838. 1447, 1565. fSymbolCnt: 731, 732, 734, 737, 739, 740, 746. gDefiningWay: 1407, 1409, 1429, 1433, 1445, fTime: 171, 619, 620. 1479, 1480, 1483, 1484, 1565. fTokens: 764, 767, 768, 769, 779, 805. *qDefinitional*: 1554, 1555, 1556. fTokensBuf: 764, 768, 769, 773, 774, 777, 784, gDefKind: 1411, 1413, 1420.786, 787, 789, 790. *gDefLabel*: 1547, 1548, 1565. $func_type: 1567, 1568.$ gDefLabId: 1477, 1479, 1548. FuncInstNr: 1606, 1607. gDefLabPos: <u>1477</u>, 1479, 1548. FuncPos: 1567, 1597, 1599, 1600, 1603, 1604, gDefPos: 1420, <u>1421</u>, 1543. 1605, 1607, 1609. Generalization: 1787, 1817, 1891. function: 8. geq: 209, 279. FunctorDefinitionObj: 1146, 1147. $get_attribute$: $\underline{666}$. FunctorDefinitionPtr: 1146, 1259, 1270, 1323, $get_index_compare_to_default$: 1842. 1371, 1445. qet_taq_kind : 659.

GetAccOptions: 99, 100.GetStdHandle: 163.GetAdjectiveCluster: 1724, 1735, 1758, 1900, GetString: 452, 467, 476. 1902, 1903. GetStructPatterns: 1858, 1870. GetArguments: 1690, 1696, 1698, 1714, 1729, GetTime: 170.1731, 1738, 1746, 1751, 1762. GetToken: 653, 654, 657, 662, 665, 666, 667, 668, GetArticleName: 77, 78, 194. 669, 764, 789, 805, 886. GetAttr: 688, 691, 692, 836, 837, 1192, 1310, GetTransfOptions: 106, 107.1322, 1323, 1324, 1325, 1326. GetVisible: 1835, 1837, 1843, 1845. GetAttrPattern: 1845, 1870, 1881, 1883. gExpandable: 1412, 1413, 1422, 1444, 1487. GetAttrPos: 1277, 1280, 1288, 1290, 1291, 1292, gExpPtr: 845, 850, 865, 1531, 1714, 1720, 1735, 1293, 1294, 1295, 1296, 1297, 1298, 1299, 1300, 1760, 1776, 1898, 1900, 1902, 1903. 1301, 1303, 1305, 1306, 1307, 1308, 1310, 1311, *qFieldsNbr*: 1508, 1509, 1510, 1513. 1313, 1314, 1316, 1318, 1319, 1320, 1321, 1323. gFormatsBase: 808, 1388.GetAttrValue: 653, 660, 666, 669, 1277, 1279. gFormatsColl: 808, 840, 1388, 1491, 1494, 1499, GetClosedSubterm: <u>1695</u>, 1714, 1716, 1720, 1760. 1503, 1507, 1509, 1510, 1513, 1575, 1589, 1598, GetConsoleMode: 163.1623, 1625, 1627, 1629, 1642, 1643. GetDirectoryName: 54. qIdentifyEqLociList: 1454, 1455, 1463, 1465. GetEnv: 66. gInference: <u>1440</u>, 1441, 1561. GetEnvironmentVariable: 67.gItemPtr: 846, 850, 858, 864, 1402, 1773, 1774, GetEnvironName: 78, 194. 1775, 1776, 1777, 1778, 1779, 1780, 1781, GetEnvStr: 65, 66, 67, 645. 1782, 1783, 1785, 1787, 1790, 1791, 1792, 1794, GetExtension: 54. 1796, 1798, 1800, 1803, 1804, 1807, 1809, 1811, GetFileExtName: 74, 75. 1813, 1818, 1832, 1833, 1834, 1835, 1837, 1838, GetFileName: 54, 73. 1842, 1843, 1845, 1847, 1849, 1851, 1853, 1856, $GetFileTime: 51, \underline{52}, \underline{624}.$ 1858, 1859, 1861, 1864, 1866, 1868, 1869, 1870, GetFuncPattern: 1847, 1870, 1881, 1906.1871, 1873, 1875, 1877, 1879, 1881, 1883, 1898, GetIdentifier: 1379, 1381, 1465, 1469, 1490, 1505, 1900, 1902, 1903, 1904, 1906, 1908, 1911, 1913, 1515, 1522, 1527, 1529, 1533, 1537, 1541, 1542, 1914, 1915, 1917, 1919. 1553, 1580, 1614, 1666. gIterativeLastFormula: 1440, 1441, 1561. GetIntAttr: 688, 692, 836, 837, 1192, 1288, 1290, gIterativeSteps: 1440, 1441, 1561, 1564. 1291, 1294, 1297, 1298, 1299, 1300, 1301, 1303, qIterPos: 1562, 1563, 1564. 1305, 1306, 1307, 1308, 1311, 1313, 1314, 1316, gLastAdjective: 1574, 1575.1319, 1323, 1324, 1325, 1326. gLastFormula: 1391, 1392, 1393, 1409, 1430, 1439, GetLocalTime: 170, 173, 175.1441, 1480, 1483, 1484, 1524, 1525, 1536, getmem: 805.1539, 1540, 1561, 1565, 1617, 1636, 1637, GetMem: 315, 329, 345, 379, 389, 396, 480, 483, 1638, 1642, 1643, 1644, 1645, 1646, 1647, 493, 567, 585, 655, 686. 1649, 1650, 1651, 1653, 1654, 1655, 1657, 1658, GetMEOptions: 102, 103.1659, 1660, 1667, 1668. GetMizFileName: 74, 75, 77. gLastTerm: <u>1407</u>, 1409, 1455, 1462, 1465, 1480, GetModePattern: <u>1838</u>, 1871, 1881. 1483, 1532, 1535, 1537, 1541, 1542, 1564, GetNames: 637, 638. 1565, 1579, 1580, 1585, 1586, 1607, 1611, GetObject: 322, 330, 335, 452, 468, 471, 474, 476. 1617, 1619, 1621, 1623, 1625, 1627, 1629, 1631, GetOptAttr: 688, 690. 1632, 1633, 1634, 1635. GetOptions: 104, 105, 194. gLastType: 1407, 1409, 1435, 1444, 1452, 1470, GetPhrase: 764, 788, 789, 792, 793. 1471, 1485, 1486, 1492, 1506, 1514, 1523, 1527, GetPredPattern: <u>1853</u>, 1870, 1881, 1883. 1530, 1571, 1582, 1587, 1589, 1590, 1591, 1615, GetPrivateVoc: 717, 741.1616, 1631, 1646, 1664, 1665. $GetPublicVoc: \underline{717}, \underline{742}.$ gLastWSBlock: <u>1382</u>, 1383, 1387, 1391, 1393, GetReferences: 1791, 1796, 1800. 1394, 1395, 1400, 1406, 1443, 1527, 1532, gets: 227, 229.1541, 1542. GetSchemeReference: 1796, 1800.gLastWSItem: 1382, 1383, 1387, 1391, 1393, 1394, 1395, 1400, 1406, 1429, 1439, 1443, 1527, $GetSortedNames: \underline{633}, \underline{634}.$

1532, 1541, 1542. $gSchemeConclusion: \underline{1428}, 1430, 1524.$ gSchemeIdNr: 1428, 1430, 1515. gLeftLoci: 1497, 1498, 1499, 1502, 1503. gLeftLociNbr: <u>1496</u>, 1498, 1499, 1502, 1503. gSchemeIdPos: 1428, 1430, 1515.gLeftLocus: 1464, 1465.gSchemeParams: 1427, 1430, 1515, 1523.gSchemePos: 1425, 1426.gLeftTermInReduction: 1453, 1455, 1465.Global variable: 36. *qSchemePremises*: 1428, 1430, 1524, 1525. gLocalScheme: 1425, 1426.gSchVarIds: 1520, 1521, 1522, 1523. $gSegmentPos\colon \ \underline{1467},\ 1468,\ 1471.$ gLocus: 1489, 1490, 1491. gMainSet: 1677, 1678.gSgmPos: 1511, 1512, 1514. gMeansPos: 1432, 1433, 1479, 1565. gSpecification: <u>1408</u>, 1409, 1444, 1445, 1485, 1486. gNewPattern: <u>1428</u>, 1430, 1455, 1457, 1463. gStartTime: <u>183</u>, 184, 191. gNewPatternPos: <u>1428</u>, 1430, 1457, 1463, 1465. gStructFields: <u>1449</u>, 1450, 1509, 1514. Gödel, Kurt: 1102. gStructFieldsSegment: 1511, 1512, 1513, 1514.gOtherwise: 1478, 1479, 1480, 1565. gStructPrefixes: 1421, 1422, 1450, 1506. gParamNbr: 1488, 1490, 1491, 1493, 1494, 1495, gSubexpPtr: 28, 844, 850, 872, 874, 875, 1607, 1498, 1499, 1501, 1502, 1503, 1505, 1507, 1509. 1671, 1688, 1689, 1690, 1691, 1692, 1693, 1695, gParams: 1449, 1450, 1490, 1491, 1493, 1494, 1696, 1698, 1702, 1703, 1705, 1706, 1708, 1709, 1495, 1498, 1499, 1501, 1502, 1503, 1504, 1711, 1713, 1714, 1716, 1718, 1719, 1720, 1721, 1505, 1507. 1722, 1724, 1725, 1727, 1733, 1734, 1735, 1737, gPartDef: <u>1482</u>, 1483, 1484. 1738, 1739, 1740, 1742, 1744, 1746, 1747, 1748, 1749, 1751, 1752, 1754, 1757, 1758, 1760, 1762, *qPartialDefs*: 1408, 1409, 1483, 1484, 1565. 1764, 1766, 1767, 1768, 1769, 1770, 1776, 1895, gPattern: <u>1428</u>, 1430, 1444, 1445, 1446, 1447, 1897, 1898, 1900, 1902, 1903. 1455, 1457, 1463, 1491, 1494, 1499, 1503. gPatternPos: 1428, 1444, 1445, 1446, 1447, 1450, gSubExpPtr: 1748, 1757, 1758. 1457, 1463, 1465, 1491, 1493, 1494, 1498, 1499, gSubItemKind: 1500, 1501, 1502, 1503, 1517, 1523.gSubItemPos: 1518, 1519, 1521, 1523. 1501, 1502, 1503, 1507. *qPremises*: 1418, 1419, 1437, 1439, 1443, 1473, *qSuchPos*: 1442, 1443, 1473. 1474, 1476, 1539, 1540. gSuchThatOcc: 1472, 1473.gPriority: 808, 836, 839, 840, 1602, 1608. gt: 210, 235, 241, 242, 246, 247, 303. GT: 648, 659, 666, 667, 668, 669. gPrivateId: 1528, 1529, 1532, 1533, 1535, 1536, 1537, 1538, 1541, 1542. gThatPos: 1438, 1439, 1443, 1474, 1476. *gPrivateIdPos*: 1528, 1529, 1532, 1533, 1535, gTHEFileNr: <u>1550</u>, 1551, 1552, 1556, 1557. 1536, 1537, 1538, 1541, 1542. gTypeList: <u>1516</u>, 1517, 1523, 1529, 1530, 1535, *qProofCnt*: 1388, 1390, 1391, 1393, 1399, 1401, 1536. 1638. gVisibleNbr: 1835, 1837, 1842, 1845, 1855, 1863. gPropertySort: <u>1412</u>, 1413, 1422, 1430, 1492. gWSTextProper: 1382, 1383, 1396.*qQualifiedSegment*: 1467, 1468, 1469, 1471. qWsTextProper: 1391, 1394, 1395, 1400, 1406, gQualifiedSegmentList: 1436, 1437, 1443, 1466,1443, 1527, 1532, 1541, 1542. gXMLHeader: 674, 694. Grabowski, Adam: 15, 715, 807, 1177. H: 176, 177, 180.Graham, Paul: 952. halt: 71, 76, 113, 123, 136, 642. Grammar, English: 1747. Halt: 72, 131, 155. Grammar, for Mizar: 1035. Halt_: 153, 155, 157. *qReconsiderList*: 1414, 1415, 1435, 1537, 1538. HasInDom: 506, 517, 520, 531, 541, 542, 543, gRedefinitions: 1387, 1397, 1423, 1424, 1429, <u>579</u>, 590, 591, <u>597</u>, 608. 1433, 1444, 1445, 1446, 1447, 1487, 1544, 1565. High: 27, 545, 1281. qResIdents: 1526, 1527.Hilbert's programme: 1100. gResPos: 1526, 1527.HowToDefine: 1126, 1146, 1147, 1323, 1407. Grow: 453, 469, 473. *I*: 38, 341, 382, 386, 389, 398, 400, 404, 405, 408, GrowLimit: 324, 332, 333, 373, 378, 391, 393, 418, <u>420</u>, <u>426</u>, <u>427</u>, <u>433</u>, <u>434</u>, <u>435</u>, <u>448</u>, <u>455</u>, <u>461</u>, 490, 499, 500, 558, 565, 569, 583, 601, 611. <u>462</u>, <u>471</u>, <u>479</u>, <u>499</u>, <u>503</u>, <u>504</u>, <u>510</u>, <u>515</u>, <u>516</u>, gScanner: 883, 886, 889, 890, 891, 892, 893. <u>519</u>, <u>520</u>, <u>521</u>, <u>524</u>, <u>527</u>, <u>532</u>, <u>534</u>, <u>535</u>, <u>537</u>,

```
<u>553, 559, 574, 575, 583, 589, 590, 592, 593,</u>
                                                               391, 393, 398, 408, 413, 418, 420, 421, 433, 473,
    <u>594</u>, <u>601</u>, <u>607</u>, <u>608</u>, <u>610</u>.
                                                               479, 490, 499, 500, 510, 517, 522, 530, 532, 538,
                                                               544, 546, 553, 558, 565, 569, 576, 583, 592, 595,
  <u>45, 75, 78, 79, 100, 103, 105, 200, 206, 224, 253,</u>
    333, 334, 337, 342, 348, 352, 363, 365, 390, 491,
                                                               596, 601, 657, 658, 659, 660, 698, 703, 737, 776,
    <u>522</u>, <u>530</u>, <u>538</u>, <u>540</u>, <u>541</u>, <u>543</u>, <u>576</u>, <u>584</u>, <u>602</u>, <u>677</u>,
                                                               779, 787, 789, 795, 796, 1025, 1029, 1186, 1189,
                                                               1192, 1326, 1374, 1391, 1399, 1505, 1513, 1585,
    679, 701, 702, 738, 740, 745, 759, 765, 793,
    1015, 1018, 1020, 1025, 1028, 1188, 1196, 1197,
                                                               1607, 1609, 1679, 1688, 1733, 1802, 1835.
    1198, 1200, 1205, 1207, 1225, 1238, 1240, 1248,
                                                          include: 1323, 1429, 1444, 1452, 1455, 1544.
    <u>1251</u>, <u>1256</u>, <u>1257</u>, <u>1260</u>, <u>1275</u>, <u>1333</u>, <u>1334</u>, <u>1336</u>,
                                                          IncorrectFormula: 1008, 1009.
    1339, 1340, 1341, 1345, 1348, 1349, 1354, 1356,
                                                          IncorrectFormulaPtr: 1008, 1295, 1636, 1637,
    1361, 1364, 1366, 1367, 1368, 1371, 1377.
                                                               1638.
ID: 648, 659, 662, 665, 666, 667, 668, 669.
                                                          IncorrectTermObj: 954, 955.
                                                          IncorrectTermPtr: 954, 1301, 1579, 1627, 1633,
Identification: 1906, 1910.
Identifier: 763, 765, 784, 884, 1381, 1542, 1545,
                                                               1634. 1635.
    1548, 1549, 1687, 1693, 1696, 1706, 1714, 1716,
                                                          Incorrect Type Obj: 966, 967.
    1720, 1740, 1760, 1774, 1783, 1792, 1794, 1796,
                                                          IncorrectTypePtr: 966, 1290, 1589, 1590.
    1805, 1807, 1809, 1811, 1813, 1818, 1835, 1843,
                                                          InCorrSentence: 1776, 1777.
    1845, 1847, 1853, 1855, 1873, 1877, 1880, 1898,
                                                          InCorrStatement: 1777, 1885.
    1904, 1906, 1911, 1915, 1917.
                                                          Index: 322, 330, 349, 352, 353, 354, 355, 356,
IdentifierName: <u>1187</u>, 1192, 1193.
                                                               358, 359, 428, 432, 506, 512.
IdentifyProcessing: 92, 96, 98.
                                                          IndexListPtr: 387, 394, 398, 409, 419.
IdentifyRegistrationObj: 1177, 1178.
                                                          IndexOf: 322, 337, 360, 387, 405, 428, 433, 452,
IdentifyRegistrationPtr: 1177, 1274, 1323, 1371,
                                                               470, 471, <u>497</u>, 503, <u>547</u>, 551, <u>552</u>, 564, <u>568</u>,
    1455.
                                                               572, 579, 589, 597, 607.
IdentLength: 775, 776, 782, 786.
                                                          IndexOfObject: 452, 474.
IdentRepr: 1193, 1202, 1206, 1222, 1223, 1239,
                                                          IndexOfStr: 423, 426, 427, 1283, 1284, 1285, 1286.
    1247, 1250, 1253, 1258, 1337, 1344, 1346, 1347,
                                                          Index1: 452, 453, 464.
    1350, 1355, 1358, 1363, 1370.
                                                          Index2: 452, 453, 464.
.idx File: 893, 1192.
                                                          InferenceKind: 1054, 1055, 1056, 1057, 1058.
if: 12.
                                                          InferenceName: 1253, 1254, 1320.
if_{-}def: 12.
                                                          infError: 1054, 1254, 1320, 1360, 1560.
if_not_def:
              12.
                                                          Infinite: 28.
IFDEF: 11, 12, 83, 86, 154, 157, 160, 161, 165,
                                                          Infinitive: 722, 723, 724, 725, 727, 728, 730, 747.
    166, 169, 170, 198, 313.
                                                          Infinitive: 1747.
IFNDEF: 12, 154, 160.
                                                          InfixFunctor: 1121, 1122, 1241, 1243, 1307,
IgnoreProof: 1802, 1803, 1833, 1920.
                                                               1309, 1365.
ikExplQualifiedSegm: 901, 906, 1018, 1205, 1293,
                                                          InfixFunctorPatternObj: 1121, 1122.
    1301, 1323, 1339, 1348.
                                                          InfixFunctorPatternPtr: 1121, 1243, 1307, 1365,
ikImplQualifiedSegm: 901, 904, 1018, 1204, 1205,
    1293, 1301, 1323, 1339, 1348.
                                                          InfixOperatorSymbol: 884, 1502, 1503, 1687, 1706,
Im: 276, 277, 279, 281, 283, 287, 289, 291, 294,
                                                               1714, 1716, 1718, 1720, 1722, 1754, 1760, 1805,
    296, 297, 301, 305.
                                                               1847, 1849, 1880, 1881.
Image: 313.
                                                          InfixTermObj: 922, 923.
implementation: 7.
                                                          InfixTermPtr: 922, 1029, 1227, 1301, 1348, 1607.
Implicitly Qualified Segment Obj: 903, 904.
                                                          info: 151, 154, 185, 198, 307, 646, 674, 807,
ImplicitlyQualifiedSegmentPtr: 903, 1010, 1017,
                                                               841, 1030, 1379, 1672.
    1018, 1194, 1204, 1205, 1277, 1292, 1328, 1338,
                                                          Info: 453, 466, 486, 487, 552, 562, 680, 681,
    1339, 1348, 1471, 1616, 1665.
                                                               684. 685.
in_AggrPattern: 1407, 1409, 1509, 1627.
                                                          InfoChar: 151.
In\_Format: 807, 836, 837.
                                                          InfoCurPos: 151.
Inc: 739.
                                                          InfoExitProc: 152.
inc: 78, 131, 181, 231, 332, 356, 373, 378, 383,
                                                          InfoFile: 151, 152, 689, 694.
```

 $\S1926$ Mizar Parser INDEX 605

```
infofile: 202, 203, 206, 208, 209, 210, 212, 215,
                                                                                                                                 1149, 1150, 1151, 1152, 1153, 1154, 1155, 1156.
         219, 222, 224, 226, 230, 233, 235, 237, 239,
                                                                                                                                 1157, 1158, 1159, 1160, 1161, 1162, 1163, 1164.
         244, 249, 251.
                                                                                                                                 1165, <u>1167</u>, 1168, <u>1169</u>, 1170, <u>1171</u>, 1172, <u>1173</u>,
                                                                                                                                 1174, <u>1175</u>, 1176, <u>1177</u>, 1178, <u>1179</u>, 1180, <u>1181</u>,
InfoInt: 151.
                                                                                                                                 1182, <u>1183</u>, 1184, 1281, 1287, 1288, 1289, 1290,
InfoNewLine: 151, 652.
                                                                                                                                 1291, 1292, 1293, 1294, 1295, 1296, 1297, 1298,
InfoPos: 151.
                                                                                                                                 1299, 1300, 1301, 1302, 1303, 1304, 1305, 1306,
InfoString: 151.
                                                                                                                                  1307, 1308, 1310, 1311, 1312, 1313, 1314, 1315,
InfoWord: 151.
                                                                                                                                 1316, 1317, 1318, 1319, 1320, 1321, 1322, 1323,
infProof: 1054, 1254, 1320, 1360, 1558.
                                                                                                                                 1324, 1325, 1326, 1378, 1383, 1385, 1386, 1395,
infSchemeJustification: 1054, 1062, 1253, 1254,
                                                                                                                                 1399, 1400, 1402, 1403, 1405, 1406, 1415, 1422,
         1320, 1360, 1560.
                                                                                                                                 1430, 1435, 1437, 1439, 1441, 1443, 1444, 1445,
infSkippedProof: 1054, 1254, 1320, 1360, 1399,
                                                                                                                                 1446, 1447, 1450, 1452, 1455, 1459, 1463, 1465,
                                                                                                                                 1466, 1468, 1469, 1471, 1474, 1476, 1483, 1484,
infStraightforwardJustification: 1054, 1060, 1254,
                                                                                                                                 1490, 1491, 1492, 1494, 1499, 1503, 1504, 1505,
         1320, 1354, 1360, 1371, 1560.
                                                                                                                                 1509, 1512, 1513, 1514, 1515, 1517, 1521, 1522,
 .ini File: 639.
                                                                                                                                 1523, 1524, 1525, 1527, 1529, 1531, 1532, 1535,
Init: 28, 312, 313, 317, 318, 322, 325, 326, 327,
                                                                                                                                 1536, 1537, 1538, 1539, 1540, 1541, 1542, 1545,
         349, 350, 364, 365, 366, 370, 371, 387, 388, 390,
                                                                                                                                 1548, 1553, 1556, 1558, 1559, 1561, 1564, 1565,
         409, 410, 423, 425, 426, 428, 429, 441, 442, 445,
                                                                                                                                 1568, 1570, 1571, 1573, 1575, 1579, 1580, 1582,
         447, 449, 450, 452, 454, 486, 488, 497, 498, 502,
                                                                                                                                 1583, 1589, 1590, 1591, 1607, 1612, 1613, 1614,
         506, 507, 508, 513, 514, 529, 548, 552, 554, 555,
                                                                                                                                 1616, 1617, 1619, 1621, 1623, 1625, 1627, 1629,
         556, <u>579</u>, 581, 588, <u>597</u>, 599, 606, <u>619</u>, 620, 625,
                                                                                                                                 1631, 1632, 1633, 1634, 1635, 1636, 1637, 1638,
         628, 630, 632, 634, 635, 636, 638, 643, 649, 650,
                                                                                                                                 1642, 1643, 1644, 1645, 1646, 1647, 1649, 1650,
         654, 662, 673, 722, 723, 731, 732, 733, 734, 736,
                                                                                                                                 1651, 1658, 1659, 1660, 1661, 1662, 1663, 1665.
         738, 743, <u>751</u>, 752, 754, 755, 757, <u>761</u>, 762, <u>764</u>,
                                                                                                                                 1666, 1667, 1668, <u>1669</u>, 1670, 1671.
         768, 773, 774, 777, 784, 786, 787, 805, 811, 812.
                                                                                                                        init\_val\_and\_end: 639.
         <u>813</u>, 814, <u>815</u>, 816, <u>817</u>, 818, <u>825</u>, 826, 827, 828
                                                                                                                        InitAccOptions: 95, 96, 100.
         829, 831, 832, 833, 834, 835, 836, 837, 853, 854,
                                                                                                                        InitCtrl: 159, 160, 162, 163.
         858, 859, 863, 864, 865, 870, 871, 872, 873, 874,
                                                                                                                        InitDisplayLine: 108, 109, 187.
         887, 898, 899, 901, 902, 903, 904, 905, 906, 907,
                                                                                                                        InitExitProc: 153, <u>158</u>, 195.
         908, \underline{909}, 910, \underline{911}, 912, \underline{916}, 917, \underline{918}, 919, \underline{920}, \underline{910}, \underline{9
                                                                                                                        InitFile: 680, 682, 684, 686, 694, 1329.
         921, 922, 923, 924, 925, 926, 927, 928, 929, 930,
                                                                                                                        initialize_definition_item: 1420, 1422.
         931, 932, 933, 934, 935, 936, 937, 938, 939, 940,
                                                                                                                        InitLibrEnv: 617, 643.
         941, 942, 943, 944, 945, 946, 947, 948, 949, 950.
                                                                                                                        InitNatFunc: 528, 529, 530, 1378.
         951, 952, 953, 954, 955, 958, 959, 960, 961, 962.
                                                                                                                        InitNatSeq: \underline{547}, \underline{548}.
         963, <u>964</u>, 965, <u>966</u>, 967, <u>970</u>, 971, <u>972</u>, 973, <u>974</u>,
                                                                                                                        InitParsing: 661, 662, 689.
         975, <u>976</u>, 977, <u>978</u>, 979, <u>980</u>, 981, <u>982</u>, 983,
                                                                                                                        InitPass: 186, 187.
         984, 985, 986, 987, 988, 989, 990, 991, 992,
                                                                                                                        InitProcessing: 192, 193.
         993, <u>994</u>, 995, <u>996</u>, 997, <u>998</u>, 999, <u>1000</u>, 1001,
                                                                                                                        InitScannerNames: <u>1188</u>, 1276, 1372.
         1002, 1003, 1004, 1005, 1006, 1007, 1008, 1009,
                                                                                                                        InitScanning: 653, 654, 662, 792, 805, 883,
         1010, 1011, 1013, 1036, 1037, 1038, 1039, 1040,
                                                                                                                                  890, 892.
         1041, <u>1042</u>, 1043, <u>1044</u>, 1045, <u>1048</u>, 1049, <u>1051</u>,
                                                                                                                        InitSorted: 387, 388, 409, 410, 425, 428, 429,
         1052, 1055, 1056, 1057, 1058, 1059, 1060, 1061,
                                                                                                                                 835, 836.
         1062, <u>1064</u>, 1065, <u>1066</u>, 1067, <u>1068</u>, 1069, <u>1070</u>,
                                                                                                                        InitSourceFile: 883, 890.
         1071, 1073, 1074, 1075, 1076, 1077, 1078, 1080.
         1081, 1083, 1084, 1086, 1087, 1089, 1090, 1091,
                                                                                                                        InitTokens: 753, 754, 756.
                                                                                                                        InitWithElement: 506, 508.
         1092, 1093, 1094, 1095, 1096, 1097, 1098, 1104,
                                                                                                                        InitWSLookupTables: 1281, 1327.
         1105, 1106, 1107, 1108, 1109, 1110, 1111, 1113,
         1114, 1115, 1116, 1117, 1118, 1119, 1120, 1121,
                                                                                                                        InitWsMizarArticle: 1379, 1383.
                                                                                                                        InOutFileBuffSize: 647, 655, 656, 686, 687,
         1122, <u>1126</u>, 1127, <u>1128</u>, 1129, <u>1130</u>, 1131, <u>1132</u>,
         1133, <u>1134</u>, 1135, <u>1136</u>, 1137, <u>1138</u>, 1139, <u>1140</u>,
                                                                                                                                 696, 699.
         1141, <u>1142</u>, 1143, <u>1144</u>, 1145, <u>1146</u>, 1147, <u>1148</u>,
                                                                                                                        Insert: 322, 332, 334, 346, 348, 349, 362, 365, 366,
```

```
370, 373, 387, 392, 409, 416, 428, 434, 452, 472,
                                                            IntPairListPtr: 486, 492, 538.
    475, 486, 490, 496, 497, 499, 506, 510, 518, 530,
                                                            IntPairSeq: 486, 487, 488, 489, 490, 491, 492,
    531, 533, 537, 538, 549, <u>552</u>, 558, 559, <u>568</u>, 569,
                                                                495, 496, 497.
    <u>579</u>, 583, 591, 592, <u>597</u>, 601, 609, 625, 628, 630,
                                                            IntPairSeqPtr: 486.
    632, 635, 636, 673, 676, 737, 738, 755, 757, 773,
                                                            IntPtr: 441, 444.
    774, 777, 784, 786, 787, 830, 831, 832, 833, 834,
                                                            IntQuickSort: 553.
    836, 887, 1281, 1287, 1289, 1293, 1297, 1301,
                                                            IntRel: 497, 498, 499, 500, 501, 502, 503, 504,
    1302, 1304, 1310, 1314, 1315, 1321, 1323, 1324,
                                                                 <u>505</u>, 506.
    1325, 1394, 1395, 1400, 1406, 1443, 1465, 1469,
                                                            IntRelPtr: 497.
    1471, 1484, 1505, 1506, 1513, 1514, 1522, 1523,
                                                            IntSequence: 552, 554, 555, 556, 557, 558, 559, 560,
    1525, 1527, 1530, 1532, 1537, 1538, 1539, 1540,
                                                                 561, 562, 563, 564, 565, 566, 567, 568, 714, 883.
    1541, 1542, 1553, 1556, 1564, 1573, 1575, 1591,
                                                            Int Sequence Ptr: 552.
    1607, 1614, 1616, 1643, 1644, 1646, 1665, 1666.
                                                            IntSet: 568, \underline{569}, \underline{570}, \underline{571}, \underline{572}, \underline{573}, \underline{574}, \underline{575},
InsertD: 428, 435.
                                                                 <u>576</u>, <u>577</u>, <u>578</u>.
InsertElem: 506, 508, 515, 518, 547, 549, 1376.
                                                            IntSetPtr: 568.
InsertExt: 370, 378, 409, 416, 418.
                                                            IntToComplex: 284, 285.
InsertHiddenFalse: 99.
                                                            IntToHex: 676, 679.
InsertHiddenFiles: 92, 96, 98, 100.
                                                            IntToStr: 43, 46, 156, 285, 710, 786.
InsertIncorrBasic: 876, 878, 1636, 1752.
                                                            IntTriplet: 485, 579, 583, 585, 588, 591, 592.
InsertIncorrFormula: 876, 878, 1637, 1744, 1748,
                                                            IntTripletList: 579.
    1757, 1758, 1760, 1776.
                                                            IntTripletListPtr: 579, 585.
InsertIncorrTerm: 876, 878, 1635, 1639, 1706,
                                                            Int2PairOfInt: 597, 601, 606, 609.
    1716, 1721.
                                                            Int2PairOfIntFunc: 597, 598, 599, 600, 601, 602,
InsertIncorrType: 876, 878, 1590, 1727, 1898.
                                                                <u>603</u>, <u>604</u>, <u>605</u>, <u>606</u>, <u>607</u>, <u>608</u>, <u>609</u>, <u>610</u>.
InsertItem: 453, 457, 472, 473.
                                                            Int2PairOfIntFuncError: 597, 598, 601, 602, 610.
InsertList: 322, 327, 334.
                                                            Int2PairOfIntFuncPtr: 597.
InsertObject: 452, 475, 476.
                                                            In WSMizFileObj: 1277, 1278, 1279, 1280, 1287,
InsertPragma: 1373, 1376.
                                                                 1288, 1289, 1290, 1291, 1292, 1293, 1294, 1295,
InsertTimes: 622, 624, 626.
                                                                 1298, 1299, 1300, 1301, 1302, 1303, 1304, 1305,
Instance: 1567, 1597, 1598, 1602, 1607, 1608, 1609.
                                                                1306, 1307, 1308, 1309, 1310, 1311, 1312, 1313,
Integer: 743, 744, 745, 807, 819, 825, 835, 1115,
                                                                <u>1314</u>, <u>1315</u>, <u>1316</u>, <u>1317</u>, <u>1318</u>, <u>1319</u>, <u>1320</u>, <u>1321</u>,
    1117, 1119, 1311, 1316, 1323, 1345, 1528, 1568.
                                                                1322, 1323, 1324, 1325, 1326.
IntegerList: 552.
                                                            In WSMizFilePtr: 1277, 1327.
IntegerListPtr: 552, 553, 567.
                                                            IODEBUG: 157.
interface: 7.
                                                            IOResult: 70, 136, 140, 153, 155, 191.
Internal Forget ful Functor Term Obj: 940, 941.
                                                            ioresult: 736.
InternalForgetfulFunctorTermPtr: 940, 1029,
                                                            IOUtils: 37.
    1232, 1301, 1348.
                                                            is\_attr\_pattern: 1880.
InternalSelectorTermObj: 934, 935.
                                                            is\_attr\_token: 1895.
                                                            is\_forgetful\_functor\_pattern: \ \ \underline{1880}.
InternalSelectorTermPtr: 934, 1029, 1194, 1224,
    1225, 1277, 1300, 1348, 1627.
                                                            is\_inference\_error: 1560.
Intersect With: 506, 517.
                                                            is\_infix\_pattern: 1880.
IntKey: 441, 442, 443.
                                                            is\_predicate\_pattern: 1880.
IntListError: 552, 561, 562, 563, 565, 566, 573.
                                                            is\_selector\_pattern: 1880.
IntPair: 441, 446, 485, 486, 490, 493, 494, 496,
                                                            is\_space: 660.
                                                            IsEqualTo: 506, 521, 568, 575.
    497, 499, 500, 502, 506, 510, 511, 513, 518,
    529, 531, 532, 538, 549, 597, 610.
                                                            IsEqWithInt: 280, 281, 293.
IntPairItem: 441, 445, 447.
                                                            IsIdentifierFirstLetter: 764, 776, 791.
IntPairItemPtr: 441, 446, 447, 448.
                                                            IsIdentifierLetter: 764, 776, 788, 791, 792, 804.
IntPairKeyCollection: 441, 446, 447, 448.
                                                            IsInSet: 568, 574, 578.
IntPairKeyCollectionPtr: \underline{441}.
                                                            IsIntegerNumber: 278, 279.
IntPairList: 486.
                                                            IsMember: \underline{497}, 504, 505.
```

IsMMLIdentifier: 75, 79. IsMultipleOf: 528, 542.IsNaturalNumber: 278, 279. IsPrime: 252, 253, 279. IsPrimeNumber: 278, 279. IsRationalGT: 282, 283.IsRationalLE: 282, 283. IsSubsetOf: 506, 522, 523, 568, 576, 577. IsSupersetOf: 506, 523, 568, 577.Is ValidSymbol: <u>718</u>, 720, 721, 728, 737. it, reserved word: 952. it_Allowed: 1407, 1409, 1481, 1483, 1484, 1633. itAssumption: 862, 1259, 1264, 1322, 1323, 1371, 1419, 1439, 1443, 1817, 1891. itAttrAntonym: 862, 1259, 1260, 1322, 1323, 1371, 1422, 1430, 1883. itAttrSynonym: 862, 1259, 1260, 1322, 1323, 1371, 1422, 1430, 1881. itAxiom: 862, 1259, 1260, 1322, 1323, 1371, 1417, 1914. itCanceled: 862, 1422, 1790. itCaseBlock: 862, 1259, 1266, 1322, 1323, 1371, 1391, 1426, 1431, 1825, 1828, 1829. itCaseHead: 862, 1259, 1264, 1322, 1323, 1371, 1417, 1439, 1828. itChoice: 862, 1259, 1260, 1322, 1323, 1371, 1417, 1437, 1805. itCluster: 862, 1259, 1273, 1322, 1323, 1371, 1422, 1452, 1898. itConclusion: 862, 1259, 1260, 1322, 1323, 1371, 1391, 1400, 1417, 1430, 1817, 1819. itConstantDefinition: 862, 1259, 1264, 1322, 1323, 1371, 1406, 1426, 1532, 1811. itCorrCond: 862, 1259, 1266, 1322, 1323, 1371, 1391, 1424, 1430, 1869. itCorrectness: 862, 1259, 1267, 1322, 1323, 1371, 1391, 1424, 1430, 1869. itDefAttr: 862, 1118, 1241, 1246, 1259, 1269, 1309, 1322, 1323, 1365, 1371, 1422, 1433, 1446, 1870. itDefFunc: 862, 1122, 1241, 1259, 1270, 1309, 1322, 1323, 1365, 1371, 1413, 1422, 1433, 1445, 1565, 1870. itDefinition: 862, 1259, 1260, 1322, 1323, 1371, 1391, 1426, 1431, 1887, 1892, 1910. itDefMode: 862, 1116, 1241, 1245, 1259, 1268, 1309, 1322, 1323, 1365, 1371, 1413, 1422, 1433, 1444, 1871. itDefPred: 862, 1120, 1241, 1242, 1259, 1269, 1281, 1309, 1322, 1323, 1365, 1371, 1413, 1422, 1433, 1447, 1870.

ItDefPred: 1285.

itDefStruct: 862, 1111, 1259, 1271, 1281, 1322, 1323, 1371, 1422, 1450, 1858. Item: 322, 339, 349, 356, 359, 360, 361, 428, 431, 435, 437, 440, 441, 443. ItemKind: 853, 858, 862, 863, 864, 1036, 1038, 1040, 1041, 1113, 1114, 1148, 1149, 1281, 1285, 1326, 1385, 1402, 1405, 1406, 1411. ItemLookUpTable: 1281.ItemLookup Table: 1281, 1282, 1285. ItemName: 1267, 1275, 1281, 1323. ItemObj: 853, 863, 864, 865, 868, 1405. ItemPtr: 850, 853, 858, 864. Items: 322, 325, 326, 327, 329, 330, 332, 333, 334, 335, 337, 341, 342, 343, 345, 346, 347, 348, 350, 354, 356, 358, 359, 363, 365, 371, 373, 378, 379, 382, 386, 388, 389, 390, 391, 393, 394, 396, 397, 400, 402, 403, 408, 412, 413, 414, 415, 418, 419, 420, 421, 427, 432, 433, 447, 448, 486, 488, 490, 491, 492, 493, 494, 499, 500, 501, 502, 507, 510, 512, 513, 514, 515, 516, 517, 521, 522, 525, 526, 529, 530, 532, 534, 535, 537, 538, 540, 541, 542, 543, 544, 545, 546, 550, 551, 624, 626, 673, 740, 747, 758, 759, 760, 766, 767, 778, 784, 790, 839, 1015, 1018, 1020, 1025, 1028, 1196, 1197, 1198, 1200, 1205, 1217, 1233, 1234, 1238, 1240, 1248, 1251, 1256, 1257, 1261, 1262, 1263, 1265, 1271, 1272, 1274, 1336, 1339, 1340, 1341, 1345, 1348, 1349, 1354, 1356, 1361, 1364, 1366, 1367, 1368, 1371, 1377, 1443, 1471, 1591, 1616, 1646, 1665, 1667, 1668. Iterative Equality Obj: 1096, 1098.IterativeEqualityPtr: 1096, 1256, 1321, 1354, 1441. IterativeStepObj: 1095, 1097.IterativeStepPtr: 1095, 1256, 1321, 1354, 1564. itExemplification: 862, 1259, 1266, 1322, 1323, 1371, 1406, 1426, 1431, 1541, 1542, 1818. itExistentialAssumption: 862, 1259, 1265, 1322, 1323, 1371, 1410, 1437, 1789. itFuncNotation: 862, 1259, 1260, 1322, 1323, 1371, 1422, 1430, 1881. itGeneralization: 862, 1259, 1264, 1322, 1323, 1371, 1406, 1410, 1430, 1817. itIdentify: 862, 1259, 1274, 1322, 1323, 1371, 1422, 1455, 1906. itIncorrItem: 862, 1260, 1285, 1322, 1323, 1371, 1805, 1881, 1883, 1885. itLociDeclaration: 862, 1259, 1264, 1322, 1323, 1371, 1406, 1422, 1430, 1891, 1894, 1910. it Mode Notation: 862, 1259, 1260, 1322, 1323,1371, 1422, 1430, 1881. itPerCases: 862, 1259, 1266, 1322, 1323, 1371, 1417, 1434, 1821.

itPragma: 862, 1259, 1260, 1322, 1323, 1371, KillBlock: 847, 849, 1822, 1830, 1887, 1892, 1395, 1431. 1910, 1915, 1921. itPredAntonym: 862, 1259, 1260, 1322, 1323, KillExpression: 845, 849, 1773, 1776, 1898, 1900, 1371, 1430, 1883. 1902, 1903, 1911. KillItem: 846, 849, 1680, 1920. itPredSynonym: 862, 1259, 1260, 1322, 1323, 1371, 1422, 1430, 1881. KillSubexpression: 844, 849, 1714, 1720, 1735, itPrivFuncDefinition: 862, 1259, 1264, 1322, 1323, 1760, 1766, 1767, 1768, 1769, 1770, 1772, 1776, 1371, 1426, 1431, 1807. 1898, 1900, 1902, 1903. itPrivPredDefinition: 862, 1259, 1264, 1322, 1323, Kind: 722, 723, 724, 727, 737, 747, 749, 752, 755, 1371, 1426, 1431, 1809. 759, 760, 762, 765, 767, 768, 773, 789, 812, 814, itProperty: 862, 1259, 1267, 1322, 1323, 1371, 816, 818, 819, 821, 836, 837, 838, 881, 886, 889, 1391, 1413, 1430, 1870. 1381, 1397, 1398, 1417, 1419, 1426, 1463, 1473, itPropertyRegistration: 862, 1259, 1274, 1322, 1491, 1501, 1502, 1507, 1517, 1542, 1543, 1544, 1323, 1371, 1391, 1422, 1455, 1908. 1545, 1546, 1548, 1549, 1555, 1556, 1557, 1558, 1559, 1560, 1561, 1572, 1575, 1588, 1626, 1628, itReconsider: 862, 1259, 1260, 1322, 1323, 1371, 1639, 1640, 1641, 1653, 1677, 1680, 1683, 1688, 1415, 1435, 1813. 1689, 1691, 1692, 1695, 1696, 1700, 1702, 1705, itReduction: 862, 1259, 1274, 1322, 1323, 1371, 1706, 1709, 1711, 1713, 1714, 1716, 1718, 1720, 1422, 1455, 1904. 1722, 1725, 1727, 1729, 1731, 1733, 1739, 1744, itRegularStatement: 862, 1259, 1260, 1322, 1323, 1746, 1747, 1748, 1751, 1752, 1754, 1756, 1757, 1371, 1391, 1417, 1430, 1805. itReservation: 862, 1259, 1260, 1322, 1323, 1371, 1758, 1760, 1762, 1766, 1767, 1768, 1769, 1770, 1774, 1779, 1790, 1791, 1792, 1794, 1796, 1798, 1406, 1426, 1431, 1527, 1911. 1800, 1802, 1803, 1804, 1805, 1813, 1815, 1816, $itSchemeBlock: \ \ \underline{862},\ 1259,\ 1260,\ 1322,\ 1323,\ 1371,$ 1391, 1426, 1431, 1915. 1817, 1818, 1819, 1820, 1821, 1822, 1823, 1827, 1828, 1830, 1832, 1838, 1840, 1843, 1845, 1847, itSchemeHead: 862, 1259, 1260, 1322, 1323, 1371, 1851, 1853, 1855, 1859, 1861, 1868, 1869, 1870, 1426, 1430, 1915. 1871, 1872, 1873, 1875, 1877, 1879, 1880, 1885, itSection: 862, 1259, 1260, 1322, 1323, 1371, 1394. 1887, 1888, 1889, 1891, 1892, 1893, 1894, 1895, itSupposeHead: 862, 1259, 1264, 1322, 1323, 1371, 1897, 1898, 1900, 1904, 1906, 1910, 1915, 1917, 1417, 1439, 1825, 1828. 1919, 1920, 1921, 1922, 1923, 1924, 1925. ItTermObj: 952, 953. $kind_is_radix_type$: 1725. ItTermPtr: 952, 1301, 1633. kn: 1606, 1607. itTheorem: 862, 1259, 1260, 1322, 1323, 1371, Knuth, Donald E.: 7. 1391, 1417, 1430, 1913. Korniłowicz, Artur: 15, 715, 807, 1177, 1181, 1183. *j*: 229, 1591, 1646. Join: <u>528</u>, 537. L: 401, 412, 432, 465, 501, 512, 571, 587, 605. l: 1584, 1597, 1609. $JoinAtom: \underline{528}, \underline{539}.$ LabelObj: 1044, 1045. Justification: 1803, 1869, 1870, 1908, 1913. JustificationObj: 1055, 1056, 1057. LabelPtr: 1044, 1045, 1089, 1091, 1092, 1126, 1127, 1130, 1131, 1134, 1135, 1194, 1247, JustificationPtr: 1055, 1093, 1094, 1095, 1096, 1277, 1310, 1311, 1312, 1321, 1328, 1350, 1399, 1097, 1098, 1158, 1159, 1160, 1161, 1162, 1163, 1405, 1545, 1547, 1548. 1164, 1165, 1181, 1194, 1252, 1254, 1266, 1277, lArg: 1306.1320, 1328, 1360, 1371, 1399, 1405, 1440, 1558. lArgs: 1307, 1308. k: 44, 346, 517, 542, 544, 546, 578, 595, 596, 755, lArgsNbr: 837. 1429, 1471, 1573, 1616, 1665, 1667, 1668. Last: 322, 331. $keep_eating_alphadigits$: <u>659</u>. lAtt: <u>690, 691</u>. $keep_getting_until_end_of_tag$: 659.*Latt*: 691. Key: 428, 432. *lAttrExp*: 28, 1895. *KeyOf*: 428, 431, 432, 433, 434, 435, 441, 443. lAttrNr: 1288. Key1: 428, 430, 437, 439, 441, 444, 446, 511, lCaseKind: 1821, 1823, 1825, 1828.622, 623. Key2: 428, 430, 437, 439, 441, 444, 446, 511, LCM: 236, 237, 528, 530. lCnt: 1188, 1189, 1192. 622, 623.

lCod: 1374.lFileExt: 73. *lCode*: 639, 718, 720, 724, 726, 1280. lFileNr: 1314.lCol: 660, 1280. *lFormat*: 826, 827, 828, 829. $\textit{lFormatNr}\colon \ \ \underline{830}, \ \underline{831}, \ \underline{832}, \ \underline{833}, \ \underline{834}, \ \underline{1491}, \ \underline{1494},$ lConds: 1323.1499, 1503, 1507, 1509, 1510, 1513, 1575, 1589, lCons: 1323.lConsoleMode: 163.1623, 1625, 1627, 1629, 1642, 1643. lFrm: 1295, 1296, 1345, 1643. lConstructorNr: 1503. $lCount: \underline{407}, 538.$ $lGcd: \underline{259}.$ $Lh: \underline{119}.$ lCounter: 1802.lCStm: 1321.lHexNr: 677, 678. lCurrChar: 772, 773, 776, 777, 779, 782, 784, LibraryPath: 627, 628. LibraryReferenceKind: 880, 882. 786, 787. lDct: 1188, 1189, 1192. Library Reference Name: 882.lDef: 1323.LibraryReferenceObj: 1050, 1051. lDefiniensExpected: 1870, 1871. Library Reference Ptr: 1050.lDefKind: 1870, 1877. librenv: 617, 674, 748, 1030. lDenom: 296, 297. lIdent: 783, 784. lDict: 736, 737.lIdNr: 1319, 1321. lDictLine: <u>728</u>, 736, 737. Limit: 322, 325, 326, 327, 329, 332, 333, 343, 347, 350, 356, 357, 365, 371, 373, 378, 379, lDictName: 743.388, 389, 390, 391, 393, 394, 397, 418, 419, *lDummy*: 738, 739, 743, 756, 757, 1188, 1189. 486, 488, 490, 492, 494, 499, 500, 502, 507, *le*: 210, 233, 269. LeadingZero: 178, 180, 182. 510, 513, 514, 529, 530, 538. Lee, Gilbert: 1100. lIndent: 1367, 1371.*lIndex*: 392, 394, 396, 419, 421, 569, 570, 778, LeftBracketName: 1187, 1189, 1190, 1191, 1228, 779, 780. 1244, 1348, 1365. Line: 128, 129, 132, 133, 138, 151, 189, 191, 193, LeftCircumfixSymbol: 884, 1501, 1503, 1687, 1700, 1706, 1714, 1716, 1720, 1760, 1805, 1847, 1880. 654, 658, 712, 768, 796, 800, 801, 846, 847, 864, 889, 1197, 1253, 1259, 1275, 1280, 1319, $lEl: \underline{652}.$ 1324, 1325, 1326, 1371. length: 38, 40, 44, 45, 68, 79, 86, 100, 103, 105, LinearReasoning: <u>1816</u>, 1822, 1830, 1920. 107, 119, 178, 200, 205, 206, 207, 212, 213, 214. lInFile: 1188, 1192, 1327. 215, 218, 219, 220, 221, 222, 223, 227, 228, 229, lInfPos: 1319.230, 398, 483, 484, 643, 657, 658, 660, 676, 677, Link: 313.678, 679, 694, 701, 719, 720, 721, 724, 725, 726, 728, 737, 742, 774, 776, 779, 780, 786, 793, *lInt*: <u>305</u>, 540, 576, <u>692</u>, 887. 795, 796, 797, 798, 799, 800, 801, 803, 1025, lIntPair: 496, 518, 531, 532, 533. 1029, 1193, 1374, 1584, 1609. lIntStr: 727. Length: 1567, 1593, 1602, 1608, 1610. $lIntTriplet: \underline{591}, \underline{592}.$ lEnvFile: 760, 836, 839.lInt2PairOfInt: 609. leq: 206, 207, 208, 209, 210, 217, 224, 226, 227, LINUX: 87. 228, 229, 231, 253, 275. ListError: 322, 330, 333, 335, 336, 352, 353, lEval: 539. 356, 357, 374, 383, 406, 408, 410, 412, 415, LexemRec: 749, 751, 753, 755, 764, 765, 811, 417, 420, 421. 836, 837. ListQuickSort: 398, 400. $lExp: \underline{1565}$. list1: 1573.lExpKind: 1310, <u>1898</u>, 1900, 1902, <u>1904</u>. list2: 1573.lExpr: 1310.lItem: 355.lExprKind: 1248. lItemKind: 1326.lFailed: 781, 782. lItems: 343, 345, 379, 394, 396, 419, 538. $lFields\colon \ \ 1323.$ lKind: 718, 719, 738, 739, 756, 757, 1188, 1189, lFieldSqmPos: 1323. 1190, 1310. *lFile*: 639, 640, 641, 743, 744. ll: 1606, 1608.

<i>lLab</i> : 1310, 1312, 1321.	LookUp_BracketFormat: <u>825</u> , 828, 831, 1623.
lLabId: 1311.	LookUp_FuncFormat: <u>825</u> , 827, 832, 1598.
lLabPos: 1311.	LookUp_PredFormat: 825, 829, 834, 1642, 1643.
<i>lLeftArgs</i> : 1606, 1607, <u>1642</u> .	LookUp_PrefixFormat: 825, 826, 833, 1575, 1589,
lLeftArgsNbr: 837, 1643.	1625, 1627, 1629.
<i>lLex</i> : 836, 837.	Look Up Func Format: 1597.
<i>lLimit</i> : 333, 538.	LOOP programming language: 1102.
<i>lLIne</i> : 742.	lOption: 107.
<i>lLine</i> : 639, 641, <u>718</u> , 719, 720, 721, <u>742</u> , <u>1280</u> .	lOtherwise: 1310.
<i>lLinked</i> : 1318.	Low: 27, 1281.
lLinkPos: 1318.	LowerCase: $43, \underline{45}$.
lList: 329, 480, 567, 1290, 1293, 1295, 1297, 1301,	<i>lPair</i> : 549.
$13\overline{10}, \ \underline{1315}, \ \underline{1323}, \ \underline{1573}.$	lpairItem: 447.
lLocus: 1323.	lPairItem: 447.
lMizarReleaseNbr: 639, 641, 642.	lParenthCnt: 1716, 1718, 1719, 1720, 1722, 1727,
lMizarVariantNbr: 639, 641, 642.	1733, 1734, <u>1760</u> , 1764.
lMizarVersionNbr: 639, 641, 642.	lPattern: 1323.
lModeMaxArgsSize: 887.	<i>lPos</i> : 639, 718, 720, 721, <u>724, 725, 726, 772, 773,</u>
lModesymbol: 1838.	774, 777, 782, 784, 786, 787, 1288, 1290, <u>1291</u> ,
lModeSymbol: 1290, 1842.	1292, 1293, 1294, 1295, 1296, 1297, 1298, 1299,
lName: 56, 58, 60, 62, <u>67</u> , 68, 625, <u>741</u> .	1300, 1301, 1303, 1305, 1306, 1307, 1308, 1310,
<i>lNbr</i> : 738, 739.	1312, 1313, 1314, 1316, 1318, 1319, 1320, 1321,
lNeg: 1345.	1323, <u>1324</u> , <u>1325</u> , 1326, <u>1679</u> .
lNoneOcc: 1288.	lPredMaxArgsSize: 887.
lNr: 756, 757, 1188, 1189, 1190, 1192, <u>1283</u> , <u>1284</u> ,	lPredSymbol: <u>1752</u> , <u>1853</u> , 1855.
1285, 1286, 1291, 1294, 1295, 1297, 1298, 1299,	lPrf: 887.
1300, 1301, 1303, 1305, 1306, 1307, 1308, 1313,	lPrintWhere: 1348.
1314, <u>1316</u> , 1319, 1323.	lPriority: 718, 720.
lNumber: 785, 786.	lProp: 1317.
Load: 722, 728, 738.	lPropertySort: 1323.
LoadDct: 753, 756, 805.	lpszTempPath: 67, 69.
LoadFIL: 622, 625.	lRedefinition: 1323.
LoadFormats: 825, 836, 1388.	lRefPtr: 1556.
	lRel: 86.
LoadMmlVcb: 717 , 743 .	
LoadPrf: 883, <u>887</u> , 892.	lRepr: 724, 725, 726.
Load Voc: 733, 738, 742, 743.	lRes: <u>261</u> , <u>263</u> , <u>265</u> , <u>267</u> , <u>269</u> , <u>285</u> , <u>287</u> , <u>289</u> , <u>291</u> ,
lObj: 1279.	293, 294, 296, 297, 436.
lObject: <u>315</u> .	lResult: 457, 458, 460, 465, 470, 474, 551, 564, 572.
LocalReference: 1047, 1049, 1250, 1251, 1314,	lRightArgs: 1606, 1607, 1642, <u>1643</u> .
1356.	lRightSymNr: 837.
LocalReferenceObj: 1048, <u>1049</u> .	<i>IRNr</i> : 1301, 1307, 1308.
LocalReferencePtr: 1048, 1194, 1250, 1277, 1313,	lRPos: 1301, 1307, 1308.
1328, 1355, 1553.	lSegment: 1616.
LocFilesCollection: 618, 628, 643.	lSels: 1323.
LociEqualityObj: 1175, 1176.	lSgm: 1295.
LociEqualityPtr: 1175, 1274, 1323, 1371, 1465.	lShape: 1310, 1326.
$LocusObj: 1108, \underline{1109}.$	lSpelling: 781, 782, 784, 786.
LocusPtr: 1108, 1117, 1118, 1175, 1176, 1194,	lStartTagNbr: <u>1326</u> .
1239, 1277, 1303, 1306, 1323, 1328, 1363, 1464,	lStr: 44, 45, 46, 67, 68, 69, 86, 87, 178, 727, 1374.
1465, 1489, 1490, 1505.	lString: 756, 757, 1188, 1189, 1190, 1192.
log_num: 198.	lStringObj: 426.
LongLines: 93, 99, 100, 103, 105, 796.	$lStruct Mode Max Args Size \colon \ \ 887.$

lStructureSymbol: 1858, 1861, 1863. MExtList.AddExtObject: 383. *lSymbNbr*: 738, 739. MExtList.DeleteAll: 375. lSymbol: 736, 737, <u>1727</u>, 1729, 1731. MExtList.DeleteExtItems: <u>385</u>. MExtList.Done: 372.LT: 648, 659, 662, 665, 666, 667, 668, 669. MExtList.FreeAll: 376.lTemp: 553.lTime: 625.MExtList.FreeExtItems: 386. lTimeStr: 180, 182. MExtList.FreeItemsFrom: 382. lToken: 755, 778, 779, 780.MExtList.Init: 371.lTrm: 1295, 1301, 1321, 1323, 1606, 1607. MExtList.Insert: 373. lType: 1323.MExtList.InsertExt: 378.MExtList.Mark: 381.lTyps: 1323.lValue: 836.MExtList.Pack: 377.lVar: 86, 1293, 1323. MExtList.SetLimit: 379.lVars: 1323.MExtListPtr: 370.lVer: 86.Meyer, Albert R.: 1102. lVer1: 639, 642. MFileExists: 47, 48, 50, 72, 627, 628, 741. lVer2: 639, 642. MFormatObj: 811, 812, 813, 838. lWSMizOutput: 1276, 1372.MFormatPtr: 807, 811, 819, 820, 821, 825, MakeEnv: 101.830, 837, 839. Mark: 370, 381. MFormatsList: 808, 825, 826, 827, 828, 829, 830, MaxArqListNbr: 1570, 1639. 831, 832, 833, 834, 835, 836, 839, 1597. $MFormatsListPtr \colon \ \ 825.$ MaxCollectionSize: <u>310</u>, 321, 344, 379, 380, 387, 395. MIndexList: 387. MaxConstInt: 748, 786. MInfixFormatObj: 815, 816, 817, 827, 829.MaxIntegerListSize: 310, 552.MInfixFormatPtr:~815,~824,~832,~834,~837,~838.MaxIntPairSize: 485, 486, 490, 492, 499. Misses: 506, 524, 525, 526, 568, 578. MaxIntTripletSize: 485, 579, 585. missing_functor_format: <u>1598</u>, 1599, 1600, 1603, MaxLineLength: 748, 796, 797. 1604, 1605. MaxListSize: 310, 452, 480, 567. Missing Word: <u>1679</u>, 1680, 1683, 1722, 1727, 1744, MaxSize: 310, 485. 1752, 1754, 1794, 1798, 1825, 1829. MaxSubTermNbr: 1379, 1388. MItemList: 321. MaxVisArgNbr: 842, 1690, 1696, 1698, 1705, MizarExitProc: 186, 190, 191, 195. 1714, 1718, 1722, 1754, 1762. MizAssert: 147, 148, 651, 670, 691, 773, 826, 836. MBracketFormatObj: 817, 818, 828. mizassert: 1193, 1391, 1406, 1591, 1646. MBracketFormatPtr: 817, 823, 831, 837, 838. Mizassert: 664, 1682. MCollection: <u>349</u>, <u>350</u>, <u>351</u>, <u>352</u>, <u>355</u>, <u>356</u>, <u>359</u>, mizenv: 35, 90, 127, 151, 154, 185, 198, 617, 674, 360, <u>361</u>, <u>362</u>, <u>363</u>, <u>364</u>, <u>365</u>, <u>366</u>, <u>367</u>, <u>368</u>, 714, 748, 879, 1030, 1373, 1379. <u>369</u>, 428, 437, 733, 764. MizExitProc: 157, 158.MCollection. AtDelete: 352.MizFileName: 36, 75, 78, 152, 193, 194, 195, MCollection.Delete: 360.643, 694, 1189, 1192. mconsole: 37, 89, 127, 151, 154, 185, 617, 748, MizFiles: 618, 628, 640, 642, 643, 644, 645, 841, 1030, 1379. 695, 760, 1195. \$MIZFILES: 617, 633, 640, 643, 645, 742, 743. *MCopy*: 312, 316, 322, 329, 334, 390. mdebug: 11. MizLoCase: 43, 45. MDEBUG: 11.MizPath: 618, 643, 645. MDuplicates: 452.MList: 322, 324, 325, 326, 327, 328, 329, 330, 331, Memory leak: 805. 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, *MError*: 186, <u>193</u>. 342, 343, 346, 347, 348, 349, 368, 369, 370, Metamathematics: 1100. 387, 389, 390, 1467, 1516, 1520, 1568. MExtList: 370, 371, 372, 373, 375, 376, 377, 378, MList.AppendTo: 346.379, 381, 382, 383, 384, 385, 386, 409. MList.At: 330.

MList.AtInsert:

MExtList.AddExtItems: 384.

$MList.CopyItems: \underline{348}.$	Move: 315, 345, 358, 379, 391, 396, 421, 463, 473,
$MList.CopyList: \underline{327}.$	480, 493, 499, 500, 502, 510, 513, 529, 561, 565,
MList.DeleteAll: 338.	567, 569, 583, 585, 588, 601, 606.
$MList.Done: \underline{328}.$	MoveCollection: 349, 364.
MList.FreeAll: 340.	MoveList: 322, 326, 349, 368, 387, 389, 1471, 1523,
MList.FreeItem: 339.	1535, 1536, 1578, 1617, 1619, 1645, 1647, 1665.
MList.FreeItemsFrom: 341.	MoveNatFunc: 528, 529.
MList. GetObject: 335.	MoveNatSet: 506, 514.
MList.ListError: 337.	$MoveObject: \overline{452}, 476.$
MList.Init: 325.	MoveSequence: 552, 556.
MList.Insert: 332.	$MoveStringList: \overline{452}, 454.$
MList.InsertList: 334.	MPrefixFormatObj: 813, 814, 815, 826.
MList.Last: 331.	MPrefixFormatPtr: 813, 822, 833, 837, 838.
MList.ListError: 336.	mscanner: 879, 1030, 1379, 1672.
MList.MCopy: 329.	MScannObj: 792, 793, 802, 803, 804, 805, 806.
MList.MoveList: 326.	MScannPtr: 792, 883, 892.
MList.Pack: 342.	Msg1: 118, 119.
MList.SetLimit: 343.	Msg2: 118, 119.
MList. TransferItems: 347.	MSortedCollection: <u>428</u> , 429, 430, <u>431</u> , <u>432</u> , 433,
MListPtr: 322.	434, 435, 437, 441, 447, 622.
<i>MML</i> : 617, 640.	MSortedCollection.Compare: 430.
mml.ini: 639.	MSortedCollection.IndexOf: 433.
MMLIdentifier: 884, 1792, 1794, 1796.	MSortedCollection.Inter. 6]: 455. MSortedCollection.Init: 429.
MMLIdentifierName: 1187, 1189, 1190, 1251,	
1253, 1356, 1359.	MSortedCollection.InitSorted: 429.
MObject: 312, 313, 314, 315, 316, 317, 322, 325,	MSortedCollection.Insert: 434.
326, 327, 329, 341, 350, 371, 382, 388, 390, 441,	MSortedExtList: 409, 410, 411, 412, 413, 414, 415,
449, 452, 454, 455, 486, 488, 507, 552, 579, 581,	<u>416, 417, 418, 419, 420, 421, 422</u> .
597, 599, 619, 653, 680, 684, 722, 731, 811, 870,	$MSortedExtList.Done: \underline{411}.$
898, 901, 907, 913, 957, 968, 1010, 1036, 1040,	$MSortedExtList.Find: \underline{412}.$
1042, 1044, 1047, 1055, 1064, 1068, 1070, 1073,	$MSortedExtList.Z: \underline{409}.$
1075, 1077, 1080, 1083, 1086, 1089, 1095, 1104,	MSortedList: <u>387</u> , 388, 389, 390, 391, 392, 394,
1106, 1108, 1110, 1126, 1128, 1132, 1136, 1142,	$400, \underline{401}, \underline{404}, 405, 407, 408, 423, 825.$
1144, 1146, 1150, 1158, 1164, 1167, 1183.	MSortedList. CopyList: 390.
mObject: 1044, 1113, 1148, 1175, 1177, 1179.	MSortedList.Find: 401.
mobjects: 307, 612, 617, 646, 674, 714, 748, 807,	$MSortedList.FreeItemsFrom: \underline{408}.$
841, 879, 895, 1030, 1373, 1379.	$MSortedList.IndexOf: \underline{405}.$
Mod: 232, 233, 235.	MSortedList.Init: 388.
ModeDefinitionObj: 1136, 1137, 1138, 1140.	MSortedList.InitSorted: 388.
ModeDefinitionPtr: 1136, 1268, 1371.	MSortedList.Insert: 392.
ModeDefinitionSort: 1136, 1137.	MSortedList.MoveList: 389.
ModeDefinitionSortName: 1268, 1323.	MSortedList.Pack: 407.
ModeMaxArgs: 883, 887, 888, 1729, 1842.	$MSortedList.Search: \overline{404}.$
ModeName: 1187, 1189, 1190, 1191, 1201, 1245,	MSortedList.SetLimit: 394.
1342, 1365.	MSortedList.Sort: 400.
ModePatternObj: 1115, <u>1116</u> .	MSortedList.Z: 391.
ModePatternPtr: 1110, 1111, 1115, 1136, 1137,	MSortedStrList: 423, 425, 426, 427, 661, 690,
1138, 1139, 1140, 1141, 1245, 1277, 1305,	753, 1281.
1323, 1365, 1444, 1494.	MSortedStrList.IndexOfStr: 426.
ModeSymbol: 884, 1588, 1589, 1706, 1725, 1727,	MSortedStrList.Init: 425.
1758, 1838, 1880, 1881, 1895.	MSortedStrList.ObjectOf: 427.
monitor: <u>153</u> , 185.	mstate: 185.
1100111001. 100.	11000000. 100.

MStringCollection: 437, 439, 440, 629, 630, nAdjectives: 976, 977, 1025, 1218, 1345. 633. 634. nAdjectiveSort: 907, 908, 1014, 1199, 1335. MStringItem: 451, 452, 455, 463, 464, 473, 480. nAdjectiveSymbol: 911, 912, 1199, 1335. MStringItemList: 452. nAllPos: 1568, 1618, 1619. MStringList: 27, 452, 454, 455, 457, 460, 461, 462, NamesFile: 630, 632, 634, 635, 638. 463, 464, 465, 466, 467, 468, 469, 470, 471, nAncestors: 1110, 1111, 1271, 1371. 472, 473, 474, 475, 476, 477, 478, 479, 480, nAntecedent: 1171, 1172, 1273, 1371. 481, 482, 717, 743, 744, 745. *nArg*: 909, 910, 930, 931, 933, 939, 982, 983, 1014, MStringList.AddObject: 460. 1025, 1029, 1117, 1118, 1199, 1208, 1230, 1231, MStringList.AddString: 457. 1246, 1335, 1345, 1348, 1365, 1643. MStringList.AddStrings: 461. nArgList: 1568, 1570, 1593, 1602, 1607, 1608, MStringList.Clear: 462. 1609, 1610, 1639. MStringList.Delete: 463. nArgListNbr: 1568, 1570, 1593, 1598, 1599, 1600, 1601, 1602, 1603, 1607, 1608, 1609, 1610, 1639. MStringList.Done: 455. MStringList.Exchange: 464. nArgs: 911, 912, 924, 925, 927, 929, 937, 958, 959, MStringList.ExchangeItems: 464. 978, 979, 1014, 1021, 1025, 1029, 1115, 1116, MStringList.Find: 465. 1117, 1118, 1121, 1122, 1199, 1201, 1206, 1223, MStringList.GetObject: 468. 1228, 1229, 1244, 1245, 1246, 1271, 1335, 1342, MStringList.GetString: 467. 1344, 1347, 1348, 1365, 1371. MStringList.Grow: 469. nArticleExt: 1036, 1037, 1196. MStringList.IndexOf: 470.nArticleId: 1196.MStringList.IndexOfObject: 474. nArticleID: 1036, 1037. MStringList.Insert: 472. nArticleNr: 1050, 1052, 1251, 1356. MStringList.InsertItem: 473. nAssumptionPos: 1150, 1151, 1272.MStringList.ObjectOf: 471. nAssumptionSort: 1150, 1151, 1272, 1362. MStringList.PutObject: 478. NatFunc: 528, 529, 530, 531, 532, 534, 535, MStringList.PutString: 477. <u>536, 537, 538, 539, 541, 542, 543, 544, 546,</u> MStringList.QuickSort: 479. 547, 611, 1373. MStringList.SetCapacity: 480. NatFuncPtr: 528, 538, 539, 540. MStringList.SetSorted: 481. $NatSeq: 547, \underline{548}, \underline{549}, \underline{550}, \underline{551}.$ MStringList.Sort: 482. NatSet: 506, 507, 508, 509, 510, 512, 513, 514, <u>515</u>, <u>516</u>, <u>517</u>, <u>518</u>, <u>519</u>, <u>520</u>, <u>521</u>, <u>522</u>, <u>523</u>, MStringList.StringListError: 466. MStringList.Z: 474. <u>524</u>, <u>527</u>, 528, 1373. *MStrObj*: <u>317</u>, 318, 426, 649, 650, 751. NatSetError: 486, 487, 490, 491, 499, 510, MStrPtr: 317, 1281. 534, 535. NatSetPtr: 506.mtime: 167, 185. MTokeniser: 764, 765, 768, 769, 773, 788, 789, nAttrCollection: 1568, 1571, 1575, 1578, 1591, 791, 792. 1646, 1647. MTokenObj: 761, 762, 764.nAttrSymbol: 1117, 1118, 1246, 1365. MTokenPtr: 761, 773, 774, 777, 784, 786, 787, 790. nAttrVals: 661, 662, 663, 671, 673, 690, 691, 1279, 1280. Mul: 27, 221, 233, 237, 248, 249, 253, 261, 263, 265, 267, 269, 275. Naumowicz, Adam: 15, 715, 807, 1177. mul: 224, 229. nBlock: 1040, 1041, 1260, 1266, 1267, 1274, 1275, MultiPredicativeFormulaObj: 974, 975. 1326, 1371, 1391. nBlockEndPos: 1036, 1039, 1197, 1325, 1393. MultiPredicativeFormulaPtr: 974, 1025, 1217, 1297, 1345, 1645. nBlockKind: 853, 854, 1036, 1039, 1197, 1367, 1386, 1391, 1393. mul1: 219, 223. Mul1: 219.nBlockPos: 1036, 1039, 1196, 1197, 1324, 1325, MUnpackTime: 176, <u>177</u>, 180. 1396. nAdjectiveCluster: 964, 965, 1021, 1201, 1342, nChoiceType: 950, 951, 1029, 1237, 1348. 1591, 1646. nClusterKind: 1167, 1168, 1273, 1371. nAdjectivePos: 907, 908, 1199. nClusterPos: 1167, 1168, 1273.

nClusterTerm: 1173, 1174, 1273, 1371. Nelson, Alexander M: great sense of humour: 778. nClusterType: 1167, 1168, 1170, 1174, 1273, 1371. nEqLociList: 1177, 1178, 1274, 1323, 1371. nConditionalDefiniensList: 1134, 1135, 1248, nEqPos: 1175, 1176, 1274.1366. New: 628, 1039. nConditions: 1086, 1087, 1154, 1155, 1157, 1162, new: 625, 673, 682, 737, 738, 741, 742, 743, 745, 1163, 1262, 1265, 1267, 1272, 1323, 1362, 1371. 755, 757, 773, 774, 777, 784, 786, 787, 831, 832, nConnective: 1568, 1653, 1658.833, 834, 836, 837, 858, 859, 865, 872, 890, 892, nConsequent: 1167, 1168, 1273, 1371. 1013, 1038, 1058, 1111, 1192, 1276, 1281, 1287, nConsistent: 528, 529, 531, 532, 534, 536, 537, 1288, 1289, 1290, 1291, 1292, 1293, 1294, 1295, 538, 539, 548. 1296, 1297, 1298, 1299, 1300, 1301, 1302, 1303, nContent: 1040, 1041, 1259, 1260, 1261, 1262, 1304, 1305, 1306, 1307, 1308, 1310, 1311, 1312, 1263, 1264, 1265, 1266, 1267, 1268, 1269, 1270, 1313, 1314, 1315, 1316, 1317, 1318, 1319, 1320, 1271, 1272, 1273, 1274, 1275, 1322, 1323, 1326, 1321, 1322, 1323, 1324, 1325, 1326, 1327, 1372, 1371, 1395, 1400, 1430, 1434, 1435, 1437, 1439, 1383, 1395, 1399, 1400, 1402, 1403, 1415, 1422, 1430, 1435, 1437, 1439, 1441, 1443, 1444, 1445, 1441, 1443, 1444, 1445, 1446, 1447, 1450, 1452, 1455, 1492, 1527, 1532, 1535, 1536, 1541, 1542. 1446, 1447, 1450, 1452, 1455, 1459, 1463, 1465, nCorrCondPos: 1158, 1159.1466, 1469, 1471, 1474, 1476, 1483, 1484, 1490, nCorrCondSort: 1160, 1161, 1259, 1371. 1491, 1492, 1494, 1499, 1503, 1504, 1505, 1509, 1512, 1513, 1514, 1515, 1522, 1523, 1524, 1525, nCurCol: 653, 654, 657, 658, 659, 660. 1527, 1531, 1532, 1535, 1536, 1537, 1538, 1539, nCurTokenKind: 653, 658, 659, 662, 664, 665, 666, 668, 669. 1540, 1541, 1542, 1545, 1548, 1553, 1556, 1558, 1559, 1561, 1564, 1565, 1573, 1575, 1578, 1579, nDefAttrPattern: 1142, 1143, 1269, 1371. 1580, 1582, 1583, 1589, 1590, 1591, 1607, 1614, *nDefAttrPos*: 1142, 1143. 1616, 1617, 1619, 1621, 1623, 1625, 1627, 1629, nDefFuncPattern: 1146, 1147, 1270, 1371. *nDefFuncPos*: 1146, 1147. 1631, 1632, 1633, 1634, 1635, 1636, 1637, 1638, 1642, 1643, 1645, 1646, 1647, 1649, 1650, 1651, nDefiniens: 1140, 1141, 1142, 1143, 1144, 1145, 1658, 1659, 1660, 1665, 1666, 1667, 1668, 1671. 1146, 1147, 1268, 1269, 1270, 1371. $new_Bl: 1597, 1604.$ nDefiniensProhibited: <u>1408</u>, 1409, 1433, 1487. NewAccom: 102, 103.nDefiningWay: 1146, 1147, 1259, 1371. NewBlock: 1036, 1038, 1391. nDefKind: 1136, 1137, 1268, 1371. NewIndex: 452, 476. nDefLabel: 1126, 1127, 1248, 1366. NewItem: 1036, 1038, 1394, 1395, 1400, 1406, nDefModePattern: 1136, 1137, 1268, 1371. 1443, 1527, 1532, 1541, 1542. nDefModePos: 1136, 1137.nDefNr: 1051, 1052, 1251.New Signal: 160.NewStr: 473, 477, 483, 611, 620, 630, 632, nDEfNr: 1356.635, 636. nDefPos: 1126, 1127, 1248. nExpansion: 1138, 1139, 1268, 1371. nDefPredPattern: 1144, 1145, 1269, 1371. nExpKind: 870, 871, 1010, 1011. nDefPredPos: 1144, 1145.nDefSort: 1126, 1127, 1248, 1366, 1429. nExpr: 1128, 1129, 1248, 1366. nExpression: 1130, 1131, 1248, 1366. nDefStructPattern: 1110, 1111, 1271, 1371. nExprKind: 1128, 1129, 1248, 1366. nDisplayInformationOnScreen: 1194, 1195, 1259, 1277, 1278, 1324, 1325, 1326, 1328, 1329, 1371. $next_term_has_higher_precedence$: 1602 NextElementState: 661, 668, 670, 836, 837, 1192, NegatedAdjectiveObj: 909, 910.NegatedAdjectivePtr: 909, 1014, 1199, 1288, 1288, 1289, 1290, 1291, 1292, 1293, 1294, 1295, 1296, 1297, 1298, 1299, 1300, 1301, 1302, 1335, 1575. NegativeFormulaObj: 982, 983. 1303, 1304, 1305, 1306, 1307, 1308, 1310, 1311, 1312, 1313, 1314, 1315, 1318, 1319, 1320, 1321, NegativeFormulaPtr: 982, 1025, 1208, 1295, 1345, 1323, 1324, 1325, 1326. 1643, 1646, 1647, 1651. nElName: 661, 662, 663, 671, 836, 1192, 1288, *NextTag*: $\underline{661}$, $\underline{665}$. nFieldPos: 1104, 1105, 1271.1290, 1293, 1295, 1301, 1307, 1309, 1310, 1311, 1314, 1315, 1320, 1321, 1323, 1324, 1325, 1326. nFields: 1106, 1107, 1271, 1371. nFieldSeqmPos: 1106, 1107, 1271. nelName: 1318.

nFieldSymbol: 1104, 1105, 1271, 1371. *nFile*: 680, 682, 683, 684, 686, 694, 696, 697, 699, 1329, 1330, 1331, 1332, 1333. nFileBuff: 684, 686, 687. nFirstSententialOperand: 1568, 1653, 1654, 1655, 1658, 1659, 1660. nFormula: 944, 945, 1029, 1233, 1348. nFormulaPos: 968, 971, 973, 975, 977, 979, 981, 1545. 983, 985, 999, 1005, 1007, 1009, 1206, 1207, 1208, 1209, 1210, 1211, 1212, 1213, 1214, 1215, 1216, 1217, 1218, 1219, 1220, 1221. nFormulaSort: 968, 971, 973, 975, 977, 979, 981, 983, 987, 989, 991, 993, 995, 997, 1001, 1003, 1005, 1007, 1025, 1207, 1343, 1345, 1643. n formula Sort: 1009.*nFunc*: 1568, 1570, 1597, 1598, 1599, 1600, 1602, 1603, 1604, 1605, 1607, 1608, 1609. nFuncId: 1073, 1074, 1264, 1371. nFunctKind: 1121, 1122, 1241, 1365. nFunctorIdent: 928, 929, 1223, 1347. nFunctorSymbol: 922, 923, 1227, 1348. nGuard: 1132, 1133, 1248, 1366. nHasAssumptions: 1385, 1387, 1410, 1424, 1475. nIdent: 898, 899, 916, 917, 1202, 1222, 1337, 1346. nIdentifier: 903, 904, 1017, 1204, 1338, 1348. nIdentifiers: 905, 906, 1018, 1070, 1071, 1205, 1257, 1339, 1348, 1371, 1616. nIdentifyPos: 1177, 1178.nIndent: 693, 694, 702, 703, 706, 708, 1328, 1329, 1334, 1367, 1371. nInDiffuse: 1385, 1388, 1391, 1542. *nInference*: <u>1405</u>, 1430, 1434, 1435, 1437, 1441, 1455, 1549, 1553, 1556, 1557, 1558, 1559, 1560, 1561, 1564. 1456. nInfPos: 1055, 1056, 1253, 1254. nInfSort: 1055, 1056, 1254, 1354, 1360, 1371, 1560. nItAllowed: 1405, 1407, 1409, 1422, 1481, 1484. nItemEndPos: 1040, 1041, 1275, 1326, 1429, 1443, 1527, 1532, 1541, 1542. nItemKind: 863, 864, 1040, 1041, 1259, 1260, 1275, 1322, 1323, 1371, 1391, 1406, 1420, 1430, 1433, 1443, 1565. nItemPos: 1040, 1041, 1259, 1275, 1371, 1405, 1409, 1430, 1437, 1439, 1452, 1455, 1492, 1532, 1541, 1542. *nItems*: 1036, 1039, 1196, 1197, 1324, 1325, 1367, 1368, 1394, 1395, 1400, 1406, 1443, 1527, 1532, 1541, 1542. nIterPos: 1095, 1097, 1256. nIterSteps: 1096, 1098, 1256, 1321, 1354.nJustification: 1080, 1081, 1086, 1087, 1093, 1094, 1095, 1097, 1158, 1159, 1164, 1165, 1181, 1182,

1255, 1256, 1262, 1263, 1266, 1267, 1274, 1321, 1323, 1352, 1354, 1371, 1455. *nLab*: 1044, 1045, 1089, 1092, 1249, 1256, 1351, 1354, 1371. nLabel: 1405, 1430, 1439, 1441, 1525, 1539,1540, 1545. nLabelIdNr: 1044, 1045, 1247, 1350, 1366, 1405, nLabelIdPos: 1405, 1545.nLabelPos: 1044, 1045, 1247. nLabId: 1048, 1049, 1250, 1355. nLastSentence: 1385, 1391, 1393. nLastWSBlock: 1385, 1387, 1393. nLastWSItem: 1385, 1387, 1393, 1394, 1395, 1405, 1406, 1430, 1434, 1435, 1437, 1441, 1443, 1444, 1445, 1446, 1447, 1450, 1452, 1455, 1492, 1527, 1532, 1535, 1536, 1541, 1542. nLeftArg: 984, 985, 1022, 1209, 1210, 1211, 1212, 1213, 1214, 1343. nLeftArgs: 922, 923, 972, 973, 1025, 1029, 1119, 1120, 1121, 1122, 1215, 1227, 1242, 1243, 1345. 1348. 1365. nLeftBracketSymb: 1121, 1122, 1244, 1365.nLeftBracketSymbol: 926, 927, 1228, 1348. nLeftLocus: 1175, 1176, 1274, 1371. nLine: 653, 654, 656, 657, 658, 659, 660. nLinkable: 1405, 1409, 1417, 1419, 1456. nLinkAllowed: 1385, 1387, 1394, 1395, 1456, 1560. nLinked: 1059, 1060, 1252, 1254, 1354, 1371, 1385, 1387, 1398, 1456, 1559, 1560, 1561. nLinkPos: 1059, 1060, 1252, 1385, 1387, 1398, 1456, 1559. nLinkProhibited: 1385, 1387, 1394, 1395, 1398, nLocusNr: 918, 919, 1029, 1226, 1348. nMizarAppearance: 1194, 1195, 1199, 1201, 1202, 1206, 1215, 1216, 1222, 1223, 1224, 1226, 1227, 1228, 1229, 1230, 1231, 1239, 1242, 1243, 1244, 1245, 1246, 1247, 1250, 1251, 1253, 1258, 1271, 1276. nModeKind: <u>1568</u>, 1588, 1589. nModeNr: 1568, 1588, 1589.nModeSymbol: 960, 961, 1115, 1116, 1201, 1245, 1271, 1342, 1365, 1371. nMultipredicateList: 1568.nMultiPredicateList: 1643, 1644, 1645. *nName*: 619, 620, 621, 623, 624, 626. nNegated: 911.nNewPattern: 1148, 1149, 1177, 1178, 1260, 1274, 1371. nNewTerm: 1183, 1184, 1274, 1371. *nNextWord*: <u>1568</u>, 1626, 1627, 1628.

nNoneOcc: 1568, 1572, 1575. 1513, 1545, 1548, 1549, 1551, 1552, 1555, 1556, nNonPos: 1568, 1572, 1575.1557, 1575, 1588, 1609, 1620, 1622, 1623, 1624, 1626, 1628, 1632, 1634, 1640, 1641, 1648, 1677, nNotationPos: 1148, 1149.nNotationSort: 1148, 1149.1729, 1731, 1747, 1748, 1756, 1757, 1794, 1798, nNotPos: <u>1568</u>, 1646, 1647, 1652. 1842, 1855, 1861, 1908. $no_longer_referencing_article$: 1794. nRedefinition: 1136, 1137, 1142, 1143, 1144, 1145, 1146, 1147, 1268, 1269, 1270, 1371. NoDisplayLine: 91, 108, 109. Noise: 112, 113, 119. nReducePos: 1183, 1184.NonBlockReasoning: 1821, 1822, 1830, 1920. nReferences: 1057, 1058, 1253, 1254, 1318, 1319, nOperSymb: 1121, 1122, 1243, 1365. 1357, 1359, 1553, 1556. nOriginPattern: 1148, 1149, 1177, 1178, 1260, nRefPos: 1047, 1049, 1052, 1250, 1251. 1274, 1371. nRefSort: 1047, 1049, 1052, 1251, 1356. nOriginTerm: 1183, 1184, 1274, 1371. nRegularStatementKind: <u>1405</u>, 1441, 1546, 1561. NotationBlock: 1892, 1923. nRestriction: 1568, 1570, 1657, 1668. $NotationDeclarationObj: 1148, \underline{1149}.$ nRestrPos: 1568, 1656, 1668. NotationDeclarationPtr: 1148, 1260, 1323, 1371, nResType: 1070, 1071, 1257, 1371. 1430. nRightArg: 984, 985, 1022, 1209, 1210, 1211, NotationsProcessing: 92, 96, 98, 99, 100. 1212, 1213, 1214, 1343. nOtherwise: 1134, 1135, 1248, 1366. nRightArgBase: 1568, 1577, 1639, 1640, 1641, NotPrivate: 1760.1642, 1643. Now: 82.nRightArgs: 922, 923, 970, 971, 973, 1025, 1029, nPartDefiniens: 1132, 1133, 1248, 1366. 1119, 1120, 1121, 1122, 1215, 1216, 1227, 1242, nPatternPos: 1113, 1114, 1242, 1243, 1244, 1243, 1345, 1348, 1365, 1643. 1245, 1246. nRightBracketSymb: 1121, 1122, 1244, 1365. nPatternSort: 1111, 1113, 1114, 1241, 1365. nRightBracketSymbol: 926, 927, 1228, 1348. *nPos*: 653, 654, 657, 658. nRightLocus: 1175, 1176, 1274, 1371. *nPostNegated*: <u>1568</u>, 1639, 1646, 1647, 1652. nRightSideOfPredPos: 1568, 1641. nPostqualification: 942, 943, 945, 1028, 1233, nRSymbolNr: 1568, 1623.1234, 1348. nSample: 942, 943, 945, 1028, 1233, 1234, 1348, nPostQualList: 1568, 1612, 1616, 1617, 1619. 1568, 1611, 1617, 1619. nPragmaStr: 1042, 1043, 1259, 1371. nSchemeConclusion: 1068, 1069, 1261, 1323, 1371. *nPredId*: 1075, 1076, 1264, 1371. nSchemeIdNr: 1061, 1062, 1068, 1069, 1253, 1258, nPredIdNr: 978, 979, 1206, 1344. 1358, 1359, 1370, 1549, 1557, 1560. nPredNr: 970, 971, 973, 1215, 1216, 1345. nSchemeInfPos: 1061, 1062, 1253, 1319, 1549, nPredSymbol: 1119, 1120, 1242, 1365. 1557. *nProp*: 1093, 1094, 1152, 1153, 1255, 1272, 1321, nSchemeParams: 1068, 1069, 1261, 1323, 1371. 1352, 1362, 1371. nSchemePos: 1068, 1069.nPropertyPos: 1164, 1165, 1179, 1180. nSchemePremises: 1068, 1069, 1261, 1323, 1371. nPropertySort: 1164, 1165, 1179, 1180, 1259, nSchFileNr: 1061, 1062, 1253, 1359, 1557. 1274, 1371. nScope: 998, 999, 1024, 1220, 1221, 1345. nPropPos: 1405, 1430, 1439, 1441, 1460, 1525, nScraps: 974, 975, 1025, 1217, 1345. 1539, 1540. nSegment: 998, 999, 1024, 1220, 1221, 1345. nQualification: 946, 947, 1029, 1235, 1348. nSegmentIdentColl: <u>1568</u>, 1613, 1614, 1616, nQualifiedSegments: 1568, 1661, 1662, 1665, 1663, 1665, 1666. 1667, 1668. nSegmentPos: <u>1568</u>, 1615, 1616, 1663, 1665. nQualVars: 1086, 1087, 1262, 1371.nSegmentSort: 901, 902, 1018, 1205, 1339, 1348. nQuaPos: 1568, 1581, 1582, 1583. nSegmPos: 901, 902, 1064, 1065, 1204, 1205, nQVars: 1156, 1157, 1265, 1323, 1371.Nr: 73, 74, 749, 752, 755, 759, 760, 762, 765, 1261, 1443, 1667, 1668. nSegmSort: 1064, 1065, 1261, 1371. 766, 767, 773, 784, 812, 814, 816, 818, 820, 836, 837, 838, 881, 886, 889, 1381, 1413, 1422, nSelectorSymbol: 932, 933, 934, 935, 1029, 1224, 1424, 1491, 1493, 1498, 1501, 1502, 1503, 1507, 1230, 1348.

nSentence: 1044, 1045, 1075, 1076, 1249, 1264, nTypePos: 957, 959, 965, 967, 1201, 1591, 1646. 1351, 1371. nTypeSort: 957, 959, 965, 967, 1021, 1201, 1342. Num: 256, 257, 259, 261, 263, 265, 267, 269, 273, nSethoodType: 1181, 1182, 1274, 1371. nSqmFields: 1110, 1111, 1271, 1371. 275, 277, 279, 281, 283, 285, 296, 305. nSntPos: 1044, 1045.numbers: $\underline{198}$, 307. nSourceFile: 653, 655, 656, 658. Numeral: 763, 765, 786, 884, 1556, 1557, 1687, 1695, 1706, 1714, 1716, 1720, 1760, 1790, nSourceFileBuff: 653, 655, 656. nSpecification: 1066, 1067, 1106, 1107, 1140, 1141, 1794, 1798, 1805. NumeralTermObj: 920, 921.1146, 1147, 1261, 1268, 1270, 1271, 1371. NumeralTermPtr: 920, 1029, 1225, 1301, 1348, nSpelling: 653, 654, 656, 657, 658, 660, 662, 671, 1632. 673, 1568, 1620, 1621, 1648, 1649. nStackArr: 1010, 1011, 1025, 1029. nValue: 649, 650, 673, 690, 691, 920, 921, 1029, nStackCnt: 1010, 1011, 1025, 1029. 1225, 1279, 1280, 1348. nState: 661, 665, 667, 668, 669, 670, 836, 1192, nVar: 1080, 1081, 1263, 1371. 1287, 1289, 1293, 1297, 1301, 1302, 1304, nVarId: 1077, 1078, 1083, 1084, 1108, 1109, 1239, 1264, 1266, 1363, 1371. 1307, 1310, 1311, 1314, 1315, 1320, 1321, nVarIdPos: 1108, 1109, 1239.1323, 1324, 1325, 1326. nVarPos: 898, 899, 1202, 1471, 1616, 1665. nStatementSort: 1089, 1090, 1092, 1098, 1256, 1259, 1354. n Vars: 1064, 1065, 1261, 1371. object: 8. nStrPos: 1110, 1111, 1271. Object: 347. nStructSymbol: 936, 937, 938, 939, 940, 941, 962, ObjectOf: 27, 423, 427, 441, 447, 452, 471, 690, 963, 1029, 1201, 1229, 1231, 1232, 1342, 1348. nSubexpPos: 1568, 1577, 1588, 1589, 1620, 1621, 691, 1279, 1280. 1626, 1627, 1628, 1629, 1630, 1631, 1639, 1640, ObjectPtr: 312. Occurs: 1683, 1693, 1708, 1737, 1739, 1740, 1778, 1642, 1643, 1645, 1646, 1648, 1649, 1653, 1658, 1782, 1783, 1785, 1787, 1804, 1811, 1813, 1818, 1661, 1662, 1667, 1668. 1837, 1842, 1843, 1845, 1861, 1864, 1866, 1873, nSubExpPos: 1647, 1658.1875, 1877, 1879, 1888, 1906, 1911, 1917. nSubject: 947, 948, 949, 976, 977, 980, 981, 1025, 1029, 1218, 1219, 1235, 1236, 1345, 1348. OldSigInt: 160.OneArgumentTermObj: 930, 931, 932, 938. nSymbolNr: 1568, 1622, 1623, 1624, 1625, 1626, 1627, 1628, 1629, 1639, 1640, 1641, 1642, 1643. One Argument Term Ptr: 930.OpenedErrors: 132, 133, 137, 142, 144. nTerm: 1095, 1097, 1256, 1354. OpenErrors: 134, 135, 193. nTermBase: <u>1568</u>, 1570, 1575, 1576, 1579, 1589, 1593, 1621, 1623, 1625, 1636, 1639, 1642, OpenFile: 688, 689, 693, 694, 695, 745, 760, $836,\ 839,\ 1192,\ \underline{1194},\ 1195,\ 1276,\ \underline{1277},\ 1278,$ 1648, 1649. 1327, 1328, 1329, 1372. nTermExpr: 1073, 1074, 1077, 1078, 1080, 1081, 1083, 1084, 1263, 1264, 1266, 1371. OpenFileWithXSL: 693, 695, 1194, 1195. nTermPos: 913, 917, 919, 921, 923, 925, 931, 935, *OpenInfoFile*: <u>151</u>, <u>152</u>, 193. OpenParenth: 1688, 1716, 1720, 1760.941, 943, 945, 947, 949, 951, 953, 955, 1222, 1223, 1224, 1225, 1226, 1227, 1228, 1229, 1230, OpenStartTag: 661, 665, 667, 669, 671. 1231, 1232, 1233, 1234, 1235, 1236, 1237. Ord: 45, 676, 679, 1281, 1285. nTermSort: 913, 917, 919, 921, 923, 925, 931, ord: 782, 786, 836, 1185, 1186, 1281, 1555, 1602, 935, 941, 943, 945, 947, 949, 951, 953, 955, 1608, 1794, 1798. 1029, 1225, 1348. Other: 517. nTheoNr: 1051, 1052, 1251, 1356. $other_regular_statements$: 1805. nType: 905, 906, 964, 965, 980, 981, 1018, 1021, othercases: 10. 1025, 1201, 1205, 1219, 1339, 1342, 1345, 1348. others: 10. nTypeChangeKind: 1080, 1081, 1263, 1371. Out_Adjective: 1194, 1199, 1200. nTypeChangeList: 1080, 1081, 1263, 1371. Out_AdjectiveList: 1194, 1200, 1201, 1218, 1273. nTypeExpList: 1064, 1065, 1073, 1074, 1075, Out_Block: 1194, 1197, 1254, 1256, 1260, 1266, 1076, 1261, 1264, 1371. 1275. nTypeExpr: 1080, 1081, 1263, 1371. Out_CompactStatement: 1194, 1255, 1256, 1260.

 $Out_Format: 811, 838, 839.$ Out_Formula: 1194, 1207, 1208, 1209, 1210, 1211, 1212, 1213, 1214, 1217, 1220, 1221, 1233, 1248, 1249, 1261, 1264. Out_ImplicitlyQualifiedVariable: 1194, 1204, 1205. $Out_InternalSelectorTerm\colon \ \underline{1194},\ 1224,\ 1225.$ $Out_Item: 1194, 1196, 1197, 1275.$ $Out_ItemContents: 1194, 1260, 1275.$ $Out_ItemContentsAttr: 1194, 1259, 1275.$ Out_Justification: 1194, 1254, 1255, 1256, 1262, 1263, 1266, 1267, 1274. Out_Label: 1194, 1247, 1248, 1249, 1256. $Out_Link: 1194, 1252, 1254.$ Out_LocalReference: 1194, 1250, 1251. Out_Loci: 1194, 1240, 1242, 1243, 1244, 1245, 1246, 1271. Out_Locus : <u>1194</u>, 1239, 1240, 1246, 1274. $Out_NumReq2$: 291. Out_NumReq3: 287, 289, 294, 297. Out_Pattern: 1194, 1241, 1260, 1268, 1269, 1270, 1274. Out_PosAsAttrs: 693, 711, 712, 1196, 1197, 1199, 1201, 1202, 1204, 1205, 1206, 1208, 1209, 1210, 1211, 1212, 1213, 1214, 1215, 1216, 1217, 1218, 1219, 1220, 1221, 1222, 1223, 1224, 1225, 1226, 1227, 1228, 1229, 1230, 1231, 1232, 1233, 1234, 1235, 1236, 1237, 1239, 1242, 1243, 1244, 1245, 1246, 1247, 1248, 1250, 1251, 1252, 1253, 1254, 1256, 1261, 1271, 1272, 1273, 1274, 1275. Out_PrivateFunctorTerm: 1194, 1223, 1225. Out_PrivatePredicativeFormula: 1194, 1206, 1207. Out_Proposition: <u>1194</u>, 1249, 1255, 1261, 1262, 1265, 1272. Out_References: <u>1194</u>, 1251, 1253, 1254. $Out_RegularStatement: 1194, 1256, 1260.$ $Out_ReservationSegment: 1194, 1257, 1260.$ Out_Reserved Variable: 1194, 1203, 1257. Out_SchemeJustification: 1194, 1253, 1254. $Out_SchemeNameInSchemeHead: 1194, 1258,$ 1261. Out_SimpleTerm: <u>1194</u>, 1222, 1225.

Out_Term: <u>1194</u>, 1198, 1218, 1219, 1225, 1230, 1231, 1233, 1234, 1235, 1236, 1248, 1256, 1263,

Out_TermList: 1194, 1198, 1199, 1201, 1206, 1215,

Out_Type: 1194, 1201, 1205, 1219, 1235, 1237,

1238, 1257, 1261, 1263, 1268, 1270, 1271,

1264, 1266, 1273, 1274.

1273, 1274.

1216, 1223, 1227, 1228, 1229.

Out_TypeList: 1194, 1238, 1261, 1264.

Out_TextProper: <u>1194</u>, 1196, 1276.

Out_Definiens: 1194, 1248, 1268, 1269, 1270.

- Out_Variable: 1194, 1202, 1203, 1204, 1205, 1261, 1263, 1264, 1266. Out_VariableSegment: <u>1194</u>, 1205, 1220, 1221, 1233, 1234, 1262, 1264, 1265. Out_XAttr: 693, 709, 710, 713, 745, 746, 747, 760, 838, 839, 1196, 1197, 1199, 1201, 1202, 1206, 1215, 1216, 1222, 1223, 1224, 1226, 1227, 1228, 1229, 1230, 1231, 1239, 1242, 1243, 1244, 1245, 1246, 1247, 1248, 1250, 1251, 1253, 1258. 1259, 1267, 1271, 1275. Out_XAttrEnd: 693, 704, 705, 745, 760, 1196, 1197, 1199, 1200, 1201, 1204, 1205, 1206, 1208, 1209, 1210, 1211, 1212, 1213, 1214, 1215, 1216, 1217, 1218, 1219, 1220, 1221, 1223, 1227, 1228, 1229, 1230, 1231, 1233, 1234, 1235, 1236, 1237, 1242, 1243, 1244, 1245, 1246, 1248, 1249, 1253, 1254, 1256, 1261, 1271, 1272, 1273, 1274, 1275. Out_XElEnd: 693, 708, 745, 747, 760, 839, 1196, 1197, 1199, 1200, 1201, 1204, 1205, 1206, 1208, 1209, 1210, 1211, 1212, 1213, 1214, 1215, 1216, 1217, 1218, 1219, 1220, 1221, 1223, 1227, 1228, 1229, 1230, 1231, 1233, 1234, 1235, 1236, 1237, 1238, 1240, 1242, 1243, 1244, 1245, 1246, 1248, 1249, 1253, 1254, 1256, 1257, 1261, 1262, 1263, 1265, 1267, 1268, 1270, 1271, 1272, 1273, 1274, 1275. Out_XElEnd0: 693, 706, 707, 711, 746, 747, 760, 838, 839, 1199, 1200, 1201, 1202, 1206, 1222, 1223, 1224, 1225, 1226, 1228, 1229, 1232, 1239, 1247, 1250, 1251, 1252, 1254, 1261, 1267, 1271. Out_XElStart: 693, 703, 705, 707, 711, 745, 746, 747, 760, 838, 839, 1196, 1197, 1199, 1200, $1201,\ 1202,\ 1204,\ 1205,\ 1206,\ 1208,\ 1209,$ 1210, 1211, 1212, 1213, 1214, 1215, 1216, 1217, 1218, 1219, 1220, 1221, 1222, 1223, 1224, 1225, 1226, 1227, 1228, 1229, 1230, 1231, 1232, 1233, 1234, 1235, 1236, 1237, 1239, 1242, 1243, 1244, 1245, 1246, 1247, 1248, 1249, 1250, 1251, 1252, 1253, 1254, 1256, 1261, 1267, 1271, 1272, 1273, 1274, 1275. Out_XElStart0: 693, 705, 745, 747, 839, 1205, 1215, 1216, 1227, 1238, 1240, 1248, 1257, 1261, 1262, 1263, 1265, 1267, 1268, 1270, 1271, 1272.
- $\begin{array}{c} 1254. \\ \textit{Out_XEl1}\colon \ \underline{693},\ 707,\ 1215,\ 1216,\ 1225,\ 1227, \\ 1240,\ 1268,\ 1269,\ 1270. \end{array}$

Out_XElWithPos: 693, 711, 1201, 1207, 1225,

 $\begin{array}{c} Out_XIntAttr\colon & \underline{693},\ 710,\ 712,\ 746,\ 747,\ 760,\ 838,\\ & 839,\ 1197,\ 1199,\ 1201,\ 1202,\ 1206,\ 1215,\ 1216,\\ & 1222,\ 1223,\ 1224,\ 1225,\ 1226,\ 1227,\ 1228,\ 1229, \end{array}$

1230, 1231, 1232, 1239, 1242, 1243, 1244, 1245, 1246, 1247, 1250, 1251, 1253, 1258, 1271, 1275.

 $\{1926$ Mizar Parser INDEX 619

 $Out_XQuotedAttr:$ 693, 713, 760. paIdentExp7: 1809, 1810. OutChar: 693, 698, 700, 701, 702, 703, 704, paIdentExp8: 1811, 1812. 708. 709. paIdentExp9: 1813, 1814.OutIndent: 693, 702, 703, 708. paIfExp: 1875, 1876, 1879. OutNewLine: 693, 700, 704, 706, 708. paInfinitiveExp: 1748, 1750, 1757. OutString: 693, 694, 695, 701, 703, 706, 708, paIsExp: 1845, 1846. paLeftCurledExp: 1917, 1918. 709, 1195. OutWSMizFileObj: 1194, 1195, 1196, 1197, 1198, paLeftDoubleExp1: 1698, 1699. 1199, 1200, 1201, 1202, 1203, 1204, 1205, 1206, paLeftDoubleExp3: 1864, 1865. 1207, 1222, 1223, 1224, 1225, 1238, 1239, 1240, paLeftParenthExp: 1807, 1808. 1241, 1247, 1248, 1249, 1250, 1251, 1252, 1253, paLeftSquareExp: 1809, 1810. <u>1254</u>, <u>1255</u>, <u>1256</u>, <u>1257</u>, <u>1258</u>, <u>1259</u>, <u>1260</u>, <u>1275</u>. paMeansExp: 1809, 1810. OutWSMizFilePtr: 1194, 1276. Panic mode: 1672. OverflowErroor: 128, 146, 155, 156. paNotExpected: 1749, 1750, 1757. OverflowError: 145, 146. paNumExp: 1794, 1795, 1798. $P: \ \underline{483}.$ paOfExp: 1709, 1710, 1908. paAdjClusterExp: 1900, 1901, 1902. paPerExp: 1823, 1824. paAllExp: 1711, 1712. paProofExp: 1830, 1831, 1920. paArrowExp1: 1856, 1857.ParamCount: 73, 74, 75, 78, 100, 103, 105, 107, paArrowExp2: 1900, 1901. 193, 634, 636, 638. paAsExp: 1813, 1814. ParamNr: 73. paAttrExp1: 1725, 1726.ParamStr: 73, 74, 75, 78, 100, 103, 105, 107, paAttrExp2: 1845, 1846. 634, 636, 638, 643. paAttrExp3: 1895, 1896. paRightBraExp1: 1700, 1701, 1705. paBegExpected: 1922.paRightBraExp2: 1851, 1852. paCasesExp: 1823, 1824. paRightCurledExp1: 1703, 1704.Pack: 322, 342, 349, 363, 370, 377, 387, 407, paRightCurledExp2: 1675.409, 417. paRightCurledExp3: 1915, 1916. paColonExp1: 1703, 1704.paRightDoubleExp1: 1698, 1699.paColonExp2: 1873, 1874, 1877. paRightDoubleExp2: 1864, 1865. paColonExp3: 1915, 1916. paRightParenthExp1: 1694, 1719, 1734.paColonExp4: 1794, 1795, 1798. paRightParenthExp10: 1722, 1723.paDefExp: 1794, 1795. paRightParenthExp11: 1845, 1846. paEndExp: 1680, 1681. paRightParenthExp2: 1696, 1697.paEqualityExp1: 1807, 1808, 1906. paRightParenthExp3: 1714, 1715. paEqualityExp2: 1811, 1812. paRightParenthExp4: 1764, 1765.paForExp: 1881, 1882, 1883, 1902, 1911. paRightParenthExp5: 1843, 1844. paForOrArrowExpected: 1898, 1899. paRightParenthExp6: 1859, 1860. paFunctExp1: 1754, 1755.paRightParenthExp7: 1796, 1797. paFunctExp2: 1849, 1850. paRightParenthExp8: 1807, 1808. paFunctExp3: 1722, 1723.paRightParenthExp9: 1917, 1918. paFunctExp4: 1898, 1904, 1905. paRightSquareExp1: 1675.paIdentExp1: 1693, 1694.paRightSquareExp2: 1762, 1763. paIdentExp10: 1873, 1874, 1877. paRightSquareExp3: 1675. paIdentExp11: 1911, 1912. paRightSquareExp4: 1809, 1810. paIdentExp12: 1675.paRightSquareExp5: 1917, 1918. paIdentExp13: 1917, 1918. Parse: 1672, 1921. paIdentExp2: 1740, 1741. $parse_comma_separated_types$: 1804. paIdentExp3: 1835, 1836, 1906. $parse_post_qualified_type$: 1692. paIdentExp4: 1783, 1784. $parse_proof: \underline{1803}.$ paIdentExp5: 1675.parser: 1379, 1672. paIdentExp6: 1807, 1808. parseraddition: 1379.

ParserOnly: 93, 105. paWrongRightBracket2: 1675. PartDefObj: 1132, 1133.paWrongSchemeVarQual: 1917, 1918. PartDefPtr: 1132, 1248, 1310, 1366, 1484. paWrongScopeBeg: 1744, 1745.paSchExp: 1798, 1799. paWrongTermBeg: 1716, 1717, 1721. paSelectExp1: 1866, 1867.PByteArray: 311.paSemicolonExp: 1680, 1681. PCharSet: 311.PassTime: 187, 189. PCMizarReleaseNbr: 81, 86, 642. paStExp: 1742, 1743. PCMizarVariantNbr: 81, 86, 642. paStillNotImplemented: 1908, 1909.PCMizarVersionNbr: 81, 86, 642. paStructExp1: 1861, 1862. PCMizarVersionStr: 84, 88, 111. paSuchExp: 1785, 1786. pcmizver: 80, 90, 185, 617, 674. paSupposeOrCaseExp: 1825, 1826, 1829. PCollection: 349.paThatExp1: 1787, 1788. PFileDescr: 619, 623, 624, 625, 626, 628. paThatExp2: 1785, 1786. PFileDescrCollection: 622. paToExp: 1904, 1905. *PFoo*: 306. PatternKindLookupTable: 1281, 1285. PIntItem: <u>441</u>, 443. PatternObj: 1113, 1114, 1115, 1117, 1119, 1121. PIntKeyCollection:441. PatternPtr: 1113, 1148, 1149, 1177, 1178, 1194, PIntSequence: 552. 1241, 1277, 1309, 1323, 1328, 1365, 1428. PIntSet: 568.paTypeExpInAdjectiveCluster: 1675. pIntSet: 568. paTypeOrAttrExp: 1758, 1759. PItemList: 321, 322, 343, 379, 394, 398, 419. paTypeUnexpInClusterRegistration: 1675. pl: 1606, 1608. paUnexpAntonym1: 1675.PlaceHolderName: 882, 1226. paUnexpAntonym2: 1675.PlaceholderTermObj: 918, 919. paUnexpConnective: 1770, 1771. PlaceholderTermPtr: 918, 1029, 1226, 1301, paUnexpEquals: 1877, 1878. 1348, 1634. paUnexpHereby: 1675.PlatformNameStr: 84, 87, 111. paUnexpItemBeg: 1885, 1886. PList: 322, 329, 905, 906, 911, 912, 922, 923, 924, paUnexpOf: 1729, 1730.925, 926, 927, 928, 929, 936, 937, 942, 943, 944, paUnexpOver: 1731, 1732. 945, 958, 959, 960, 961, 962, 963, 964, 965, 970, paUnexpReconsider: 1675. 971, 972, 973, 974, 975, 976, 977, 978, 979, paUnexpRedef: 1889, 1890. 1010, 1015, 1020, 1036, 1039, 1057, 1064, 1065, paUnexpSynonym: 1675.1066, 1067, 1068, 1069, 1070, 1071, 1073, 1075, paUnpairedSymbol: 1680, 1681. 1080, 1081, 1086, 1087, 1096, 1098, 1106, 1107, paWithExp: 1906, 1907. 1110, 1111, 1115, 1116, 1117, 1118, 1119, 1120, paWrongAfterThe: 1706, 1707.1121, 1122, 1134, 1135, 1154, 1155, 1156, 1157, paWrongAttrArgumentSuffix: 1675. 1167, 1168, 1169, 1170, 1171, 1172, 1173, 1174, paWrongAttrPrefixExpr: 1675. 1177, 1178, 1194, 1198, 1200, 1238, 1240, 1251, paWrongFormulaBeg: 1760, 1761. 1277, 1287, 1289, 1293, 1295, 1297, 1301, 1302, paWrongFunctorPatternBeg: 1847, 1848. 1304, 1307, 1308, 1310, 1314, 1315, 1321, 1323, paWrongItemBeg: 1805, 1806, 1885. 1328, 1336, 1340, 1341, 1349, 1356, 1361, 1364, paWrongJustificationBeq: 1800, 1801. 1379, 1408, 1414, 1415, 1418, 1421, 1422, 1427, $pa Wrong Mode Pattern Beg: \ \ 1838, \ 1839.$ 1428, 1436, 1440, 1449, 1451, 1454, 1458, 1459, paWrongModePatternSet: 1840, 1841. 1463, 1466, 1471, 1474, 1476, 1483, 1497, 1504, paWrongPattBeg1: 1881, 1882. 1509, 1511, 1515, 1523, 1526, 1535, 1536, 1539, paWrongPattBeg2: 1883, 1884. 1561, 1573, 1578, 1591, 1606, 1607, 1616, 1617, paWrongPattBeg3: 1675.1619, 1642, 1643, 1645, 1646, 1647, 1665. paWrongPredPattern: 1853, 1854. Plist: 1058, 1073, 1074, 1075, 1076, 1289, 1290, paWrongPredSymbol: 1752, 1753. 1310, 1314, 1512, 1524, 1527. paWrongRadTypeBeg: 1727, 1728.PObject: 312, 314, 315, 316, 322, 329, 334, 339, paWrongReferenceBeg: 1792, 1793, 1796. 348, 349, 366, 382, 386, 390, 408, 420, 423, 427, paWrongRightBracket1: 1675. 435, 451, 452, 460, 468, 471, 474, 475, 476, 478,

1040, 1128, 1279, 1310, 1478, 1482, 1591, 1646. PredicatePatternObj: 1119, 1120.Pobject: 1129.PredicatePatternPtr: 1119, 1144, 1145, 1242, 1277, 1308, 1323, 1365, 1447, 1499. Pointer: 320. PredicateSegment: 1064, 1261, 1323, 1371, 1523. Pos: 128, 130, 131, 132, 133, 151, 193, 639, 718, PredicateSegmentPtr: 1066, 1261, 1323, 1371. 720, 721, 724, 774, 800, 801. PredicateSymbol: 884, 1640, 1641, 1747, 1748, Position: 128, 130, 131, 132, 133, 151, 193, 653, 1752, 1756, 1757, 1760, 1805, 1853, 1880, 693, 711, 712, 761, 762, 764, 772, 883, 898, 899, 901, 902, 903, 904, 905, 906, 907, 908, 909, 910, 1881, 1883. PredicativeFormulaObj: 972, 973. 911, 912, 913, 916, 917, 918, 919, 920, 921, 922, PredicativeFormulaPtr: 972, 1025, 1215, 1297, 923, 924, 925, 926, 927, 928, 929, 930, 931, 932, 1345, 1642. 933, 934, 935, 936, 937, 938, 939, 940, 941, 942, PredMaxArgs: 883, 887, 888, 1746, 1751, 1855. 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, PrefixFormatChars: 826. 953, 954, 955, 957, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967, 968, 970, 971, 972, 973, 974, Prel directory: 625. 975, 976, 977, 978, 979, 980, 981, 982, 983, 984, prephan: 198. 985, 986, 987, 988, 989, 990, 991, 992, 993, 994, Previous: 449, 854, 856, 864, 874, 875, 1391. 995, 996, 997, 998, 999, 1000, 1001, 1002, 1003, PrevPos: 883, 886, 1429, 1443, 1465, 1527, 1532, 1004, 1005, 1006, 1007, 1008, 1009, 1036, 1037, 1541, 1542, 1557, 1559, 1679, 1889, 1915. 1038, 1039, 1040, 1041, 1044, 1045, 1047, 1048, PrevWord: <u>883</u>, 886. 1049, 1051, 1052, 1055, 1056, 1057, 1058, 1059, .prf File: 887. 1060, 1061, 1062, 1064, 1065, 1066, 1067, 1068, Primitive Recursive Function: 1102. 1069, 1095, 1097, 1104, 1105, 1106, 1107, 1108, Print_Adjective: <u>1328</u>, 1335, 1336. 1109, 1110, 1111, 1113, 1114, 1115, 1116, 1117, Print_AdjectiveList: 1328, 1336, 1342, 1345, 1371. 1118, 1119, 1120, 1121, 1122, 1126, 1127, 1130, $print_arguments$: 1201. 1131, 1134, 1135, 1136, 1137, 1138, 1139, 1140, Print_AssumptionConditions: <u>1328</u>, 1362, 1371. 1141, 1142, 1143, 1144, 1145, 1146, 1147, 1148, Print_BinaryFormula: 1328, 1343, 1345. 1149, 1150, 1151, 1152, 1153, 1154, 1155, 1156, Print_Block: 1328, 1354, 1360, 1367, 1371. 1157, 1158, 1159, 1160, 1161, 1162, 1163, 1164, Print_Char: 1328, 1330, 1332, 1333, 1334, 1348. 1165, 1167, 1168, 1169, 1170, 1171, 1172, 1173, $Print_CompactStatement: 1328, 1352, 1354.$ 1174, 1175, 1176, 1177, 1178, 1179, 1180, 1181, Print_Conditions: 1328, 1361, 1362, 1371. 1182, 1183, 1184, 1277, 1280, 1288, 1290, 1291, Print_Definiens: 1328, 1366, 1371. 1292, 1293, 1294, 1295, 1298, 1299, 1300, 1301, Print_Formula: <u>1328</u>, 1343, 1345, 1348, 1351, 1303, 1305, 1306, 1307, 1308, 1310, 1311, 1312, 1366, 1371. 1313, 1314, 1316, 1318, 1319, 1320, 1321, 1323, $Print_Implicitly Qualified Variable: \underline{1328}, 1338,$ 1324, 1325, 1326, 1385, 1405, 1421, 1425, 1428, 1339. 1432, 1438, 1442, 1467, 1477, 1511, 1518, 1526, Print_Indent: <u>1328</u>, 1334, 1367, 1371. 1528, 1562, 1567, 1568, 1597, 1677, 1679, 1680, Print_Item: 1328, 1367, 1368, 1371. 1802, 1821, 1830, 1887, 1892, 1910, 1915. Print_Justification: 1328, 1352, 1354, 1360, 1371. POSIX: 65, 128. Print_Label: 1328, 1350, 1351, 1354, 1366, 1371. Pragma: 774. Print_Linkage: <u>1328</u>, 1353, 1354, 1371. Pragma: 884, 1815, 1888, 1893, 1910, 1924. Print_Loci: 1328, 1364, 1365, 1371. Pragma, ::\$E0F: 774. Print_Locus: 1328, 1363, 1364, 1365, 1371. PragmaObj: 1042, 1043. Print_NewLine: <u>1328</u>, 1331, 1354, 1361, 1366, PragmaPtr: 1042, 1259, 1322, 1371, 1395. 1367, 1371. pragmas: 1373, 1379, 1672. $Print_Newline: 1371.$ pred: 334, 337, 354, 363, 389. Print_Number: <u>1328</u>, 1332, 1348, 1356, 1359. Predicate: 1752. Print_OpenTermList: 1328, 1340, 1342, 1344, PredicateDefinitionObj: 1144, 1145. 1345, 1347, 1348. PredicateDefinitionPtr: 1144, 1269, 1323, 1371, Print_Pattern: 1328, 1365, 1371. 1447. Print_PrivateFunctorTerm: 1328, 1347, 1348. PredicateName: 1187, 1189, 1190, 1191, 1215, Print_PrivatePredicativeFormula: 1328, 1344, 1216, 1242, 1345, 1365. 1345.

Print_Proposition: 1328, 1351, 1352, 1361, 1362, Process_FinishQuantifiedFormula: 1010, 1023, 1025. 1371. Process_FinishVariableSegment: 1010, 1018, 1019. Print_Reference: <u>1328</u>, 1355, 1356. Process_Formula: 1010, 1022, 1024, 1025, 1027. Print_References: <u>1328</u>, 1356, 1357, 1359. $Process_FraenkelTermsScope: 1010, 1027, 1029.$ Print_RegularStatement: 1328, 1354, 1371. Process_ImplicitlyQualifiedVariable: 1010, 1017, Print_ReservedType: 1328, 1369, 1371. 1018. Print_SchemeJustification: <u>1328</u>, 1359, 1360. $process_notation_item: 1457.$ Print_SchemeNameInJustification: 1328, 1358, $Process_QuantifiedFormula: 1010, 1024, 1025.$ 1359. $Process_SimpleFraenkelTerm: 1010, 1028, 1029.$ Print_SchemeNameInSchemeHead: 1328, 1370, Process_SimpleTerm: 1010, 1026, 1029. 1371. Process_StartFraenkelTerm: 1010, 1026, 1029. *Print_SimpleTermTerm*: <u>1328</u>, 1346, 1348. Process_StartQuantifiedFormula: 1010, 1023, Print_StraightforwardJustification: 1328, 1357, 1025. 1360. Process_StartVariableSegment: 1010, 1018, 1019. Print_String: 1328, 1333, 1335, 1337, 1339, 1340, Process_Term: 1010, 1020, 1025, 1028, 1029. 1341, 1342, 1343, 1344, 1345, 1346, 1347, Process_TermList: 1010, 1014, 1020, 1021, 1025, 1348, 1349, 1350, 1353, 1354, 1355, 1356, 1029. 1357, 1358, 1359, 1361, 1363, 1364, 1365, *Process_Type*: <u>1010</u>, 1018, 1021, 1025, 1029. 1366, 1367, 1370, 1371. Process_Variable: <u>1010</u>, 1016, 1017, 1018. Print_Term: <u>1328</u>, 1340, 1341, 1345, 1348, 1354, $Process_VariablesSegment: 1010, 1018, 1024, 1028.$ 1366, 1371. ProcessArguments: <u>1716</u>, 1725, 1897. Print_TermList: 1328, 1335, 1341, 1348. ProcessAtomicFormula: 876, 878, 1639, 1752, Print_TextProper: <u>1328</u>, 1368, 1372. 1758. Print_Type: 1328, 1339, 1342, 1345, 1348, 1349, ProcessAttrAntonym: 866, 868, 1457, 1883. 1369, 1371. ProcessAttribute: 876, 878, 1575, 1725, 1895. Print_TypeList: <u>1328</u>, 1349, 1371. ProcessAttributeName: 661, 669, 673. Print_Variable: 1328, 1337, 1338, 1339, 1348, ProcessAttributePattern: 867, 868, 1845. 1371. ProcessAttributes: 1724, 1725. Print_VariableSegment: <u>1328</u>, 1339, 1345, 1371. ProcessAttributeValue: 661, 669, 673. $Print_WSMizArticle: 1372.$ ProcessAttrIdentify: 867, 868. Prior: 722, 723, 724, 726, 727, 728, 747. ProcessAttrSynonym: 866, 868, 1457, 1881. private: 653.ProcessBegin: 853, 857, 1385, 1394, 1922, 1924. PrivateFunctorDefinitionObj: 1073, 1074. ProcessBeing: 866, 868, 1470, 1783. PrivateFunctorDefinitionPtr: 1073, 1264, 1323, ProcessBinaryConnective: <u>876</u>, 878, 1653, 1766, 1371, 1535. 1768, 1770. PrivateFunctorTermObj: 928, 929.ProcessCase: 1822, 1825, 1828, 1829. PrivateFunctorTermPtr: 928, 1029, 1194, 1223, ProcessChoice: <u>1785</u>, 1789, 1805. 1225, 1277, 1299, 1328, 1347, 1348, 1621. ProcessComment: 792, 800, 801, 802. PrivateItem: 1805, 1817, 1820, 1885, 1891, ProcessContradiction: <u>876</u>, 878, 1650, 1760. 1910, 1923. Process Correctness: 866, 868, 1543, 1869. PrivatePredicateDefinitionObj: 1075, 1076. ProcessDef: 866, 868, 1555, 1794. PrivatePredicateDefinitionPtr: 1075, 1264, 1323, ProcessDefiniensLabel: 866, 868, 1548, 1873, 1877. 1371, 1536. ProcessDoesNot: 877, 878, 1749, 1757. PrivatePredicativeFormulaObj: 978, 979. ProcessEndTag: 661, 668, 672. PrivatePredicativeFormulaPtr: 978, 1025, 1194, ProcessEquals: 866, 868, 1479, 1877. 1206, 1207, 1277, 1294, 1328, 1344, 1345, 1649. ProcessExactly: 876, 878, 1583, 1689. Procedure: 1672.ProcessExemplifying Variable: 866, 868, 1541, Process_Adjective: <u>1010</u>, 1014, 1015. 1818. Process_AdjectiveList: <u>1010</u>, 1015, 1021, 1025. ProcessField: 866, 868, 1513, 1866. Process_BinaryFormula: 1010, 1022, 1025. ProcessFixedVariable: 866, 868, 1469, 1783.

ProcessFlexConjunction: <u>876</u>, 878, 1655, 1767.

Process_FinishFraenkelTerm: 1010, 1026, 1029.

ProcessFlexDisjunction: 876, 878, 1654, 1769. ProcessSchemeName: 866, 868, 1515, 1915. ProcessFuncIdentify: 866, 868, 1463, 1906. ProcessSchemeNumber: 866, 868, 1557, 1798. ProcessFuncReduction: 866, 868, 1465, 1904. *ProcessSchemeReference*: <u>866</u>, 868, 1549, 1796. ProcessFuncSynonym: 866, 868, 1457, 1881. ProcessScheme Variable: <u>866</u>, 868, 1522, 1917. ProcessFunctorSymbol: 866, 868, 876, 878, 1502, ProcessSentence: 1775, 1778, 1781, 1785, 1832, 1609, 1714, 1849. 1913, 1914, 1919. ProcessHolds: 877, 878, 1744. ProcessSimpleTerm: <u>876</u>, 878, 1580, 1696. *ProcessHypotheses*: <u>1778</u>, 1780, 1787. ProcessStructureSymbol: <u>866</u>, 868, 1507, 1861. ProcessingEnding: 196, 197.ProcessThe: 877, 878, 1706. ProcessIterativeStep: 867, 868, 1834. ProcessTheoremNumber: 866, 868, 1556, 1794. ProcessItTerm: 876, 878, 1633, 1695. Process Thesis: 876, 878, 1638, 1760. ProcessLab: <u>1774</u>, 1778, 1781, 1785, 1832, 1913, Process Variable: 876, 878, 1666, 1740. 1914, 1919. Process Visible: 866, 868, 1505, 1835. ProcessLabel: 866, 868, 1545, 1774, 1777. program: 7. ProcessLeftLocus: 866, 868, 1465, 1906. *Prohibited*: 793, 798. ProcessLeftParenthesis: 877, 878, 1688, 1714, ProofPragma: 1373, 1375, 1399, 1558, 1803, 1733. 1833, 1920. ProcessLink: 853, 857, 1385, 1398, 1805, 1819, PropertiesProcessing: 92, 96, 98. 1820. PropertyKind: 880, 882, 1164, 1165, 1179, 1180, ProcessLocusTerm: 876, 878, 1634, 1695. 1181, 1182, 1281, 1286, 1323, 1412, 1413, ProcessMeans: 866, 868, 1479, 1873. 1422, 1908. ProcessModePattern: 867, 868, 1842. PropertyKindLookupTable: 1281, 1282, 1286. ProcessModeSymbol: 876, 878, 1588, 1727. PropertyName: 882, 1259, 1281, 1371. ProcessModeSynonym: 866, 868, 1457, 1881. PropertyObj: 1164, 1165. ProcessNegation: 876, 878, 1652, 1758. PropertyPtr: 1164, 1259, 1267, 1323, 1371, 1430. ProcessNegative: 876, 878, 1651, 1748, 1757, 1760. PropertyRegistrationObj: 1179, 1180, 1181. ProcessNon: 876, 878, 1572, 1725, 1895. PropertyRegistrationPtr: 1179, 1274, 1371. ProcessNot: 877, 878, 1760. Proposition Obj: 1044, 1045.ProcessNumeralTerm: <u>876</u>, 878, 1632, 1695. PropositionPtr: 1044, 1093, 1094, 1096, 1098, ProcessPostqualification: <u>1691</u>, 1703, 1711. 1152, 1153, 1194, 1249, 1277, 1312, 1317, 1328, ProcessPostqualifiedVariable: 876, 878, 1614, 1351, 1399, 1430, 1439, 1441, 1525, 1539, 1540. Prune: 349, 367. ProcessPragma: 853, 857, 1385, <u>1395</u>, 1815. $PSortedCollection: \underline{428}.$ ProcessPragmas: <u>1815</u>, 1816, 1888, 1893, 1910, *PStr*: 317, 424. 1924. PString: 311, 439, 451, 483, 484, 611, 619. ProcessPredAntonym: 866, 868, 1457, 1883. PStringCollection: 437. ProcessPredicateSymbol: 866, 868, 876, 878, 1498, PStringItem: 451.1640, 1751, 1853. $PStringItemList: \underline{452}, 480.$ ProcessPredIdentify: 867, 868. $PStringList: \underline{452}.$ ProcessPredSynonym: <u>866</u>, 868, 1457, 1881. PSymbol: 722, 736, 737, 738, 740, 747. ProcessPrivateReference: 866, 868, 1553, 1792. PublicLibr: 106, 107.ProcessQua: 876, 878, 1581, 1689. $PUsortedStringCollection: \underline{437}.$ ProcessReconsideredVariable: 866, 868, 1537, PutError: 128, 130, 131, 134, 135, 191, 193. PutObject: 452, 460, 475, 478. ProcessRedefine: 853, 857, 1385, <u>1397</u>, 1888. PutString: 452, 477. ProcessReservedIdentifier: 866, 868, 1527, 1911. PVocabulary: 717, 733, 741, 742, 743, 744, 745. ProcessRightLocus: 866, 868, 1465, 1906. PWordArray: 311. ProcessRightParenthesis: 877, 878, 1688, 1714, *P2*: **27**. 1718, 1719, 1722, 1734, 1754, 1764. *P3*: **27**. ProcessRightSideOfPredicateSymbol: 876, 878, 1641, 1746. QT: 648, 659, 666, 669. ProcessSch: 867, 868, 1798. $\label{eq:QualifiedSegmentObj: 901, 902, 903, 905.} QualifiedSegmentObj: 901, 902, 903, 905.$

QualifiedSegmentPtr: 901, 998, 999, 1000, 1001, Read_Formula: 1277, 1295, 1296, 1297, 1301, 1002, 1003, 1010, 1018, 1028, 1194, 1205, 1310, 1312, 1323. $1264,\ 1277,\ 1293,\ 1295,\ 1328,\ 1339,\ 1348,$ $Read_FunctorPattern\colon \ \underline{1277},\ 1307,\ 1309,\ 1323.$ 1371, 1443, 1667, 1668. Read_ImplicitlyQualifiedSegment: 1277, 1292, Qualified TermObj: 946, 947. 1293. Qualified TermPtr: 946, 1029, 1235, 1301, 1348, Read_InternalSelectorTerm: 1277, 1300, 1301. Read_Item: 1277, 1324, 1325, 1326. 1582. $Read_ItemContents: 1277, 1323, 1326.$ QualifyingFormulaObj: 980, 981. $Read_ItemContentsAttr: 1277, 1322, 1326.$ QualifyingFormulaPtr: 980, 1025, 1219, 1295, Read_Justification: <u>1277</u>, 1317, 1320, 1321, 1323. 1345, 1646. $Read_label$: 1312. QuantifiedFormulaObj: 998, 999, 1000, 1002.Read_Label: 1277, 1310, 1311, 1321. QuantifiedFormulaPtr: 998, 1010, 1023, 1024, *Read_LocalReference*: 1277, 1313, 1314. 1025, 1220, 1221, 1345. Read_Loci: 1277, 1304, 1305, 1306, 1307, 1308, Quantified Variables: 1739, 1742, 1744. 1323. QuickSort: 453, 479, 482. Read_Locus: 1277, 1303, 1304, 1306, 1323. Quicksort: <u>398</u>, 399, 479, 482, 553. Read_ModePattern: <u>1277</u>, 1305, 1309, 1323. QuietMode: 93, 99, 100, 103, 105, 117, 189. Read_Pattern: <u>1277</u>, 1309, 1323. QuoteStrForXML: 674, 676, 713, 747, 1190, $Read_PredicatePattern: 1277, 1308, 1309, 1323.$ 1226, 1259. Read_PrivateFunctorTerm: 1277, 1299, 1301. QuoteXMLAttr: 674, 679. Read_PrivatePredicativeFormula: 1277, 1294, $r\colon \ \ \underline{233}, \ \underline{239}, \ \underline{244}, \ \underline{249}, \ \underline{251}, \ \underline{255}, \ \underline{258}, \ \underline{259}.$ 1297. Radix type: 1727. Read_Proposition: 1277, 1312, 1317, 1323. RadixTypeObj: 958, 959, 960, 962. Read_References: <u>1277</u>, 1314, 1318, 1319. RadixTypePtr: 958. $Read_RegularStatement: 1277, 1321, 1323.$ RadixTypeSubexpression: <u>1727</u>, 1735, 1758, 1900, $Read_ReservationSegment: 1277, 1315, 1323.$ 1902, 1903. Read_SchemeJustification: 1277, 1319, 1320. Rational: 256, 257, 258, 259, 260, 261, 262, 263, Read_SchemeNameInSchemeHead: 1277, 1316, 264, 265, 266, 267, 268, 269, 270, 271, 272, 1323. 273, 274, 275, 276, 296, 300, 301. Read_SimpleTerm: 1277, 1298, 1301. RationalAdd: 260, 261, 287, 294, 296, 297, 301. Read_StraightforwardJustification: 1277, 1318, RationalDiv: 270, 271, 297. 1320. RationalEq: 272, 273, 281.Read_Term: <u>1277</u>, 1287, 1295, 1301, 1310, 1321, RationalGT: 274, 275, 283.1323. RationalInv: 268, 269, 271. Read_TermList: 1277, 1287, 1288, 1290, 1294, RationalLE: 274, 275, 283. 1297, 1299, 1301. RationalMult: 266, 267, 271, 294, 296, 297, 301. Read_TextProper: 1277, 1324, 1327. RationalNeg: 264, 265, 291. Read_Type: 1277, 1290, 1293, 1295, 1301, 1302, RationalReduce: 258, 259, 261, 263, 267. 1315. 1323. RationalSub: 262, 263, 289, 294, 297. $Read_TypeList: 1277, 1302, 1323.$ RComplex: 276, 277, 278, 279, 280, 281, 282, 283, Read_Variable: 1277, 1291, 1292, 1293, 1315, 1323. 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, Read_VariableSegment: <u>1277</u>, 1293, 1295, 1301, 295, 296, 298, 299, 300, 301, 304, 305. 1323. Re: 276, 277, 279, 281, 283, 285, 287, 289, 291, $Read_WSMizArticle$: 1327. 294, 296, 297, 301, 305. readln: 737, 742, 757, 796, 1189, 1192. Read: 739, 887. ReadLn: 625, 630, 632, 635, 641, 658, 728, $Read_Adjective: 1277, 1288, 1289.$ 738, 743. Read_AdjectiveList: 1277, 1289, 1290, 1295, 1323. ReadNames: 631, 632.Read_AttributePattern: <u>1277</u>, 1306, 1309, 1323. ReadParams: 1843, 1849. ReadPrivateVoc: 733, 736, 741. Read_Block: <u>1277</u>, 1325, 1326. Read_CompactStatement: 1277, 1317, 1321, 1323. ReadSortedNames: 629, 630. Read_Definiens: <u>1277</u>, 1310, 1323. ReadToken: 883, 886, 1672.

ReadTokenProc: 1672, 1677, 1680, 1682, 1802, $req_info: 198.$ 1815, 1830, 1921, 1924, 1925. Reservation: 1911, 1923. ReadTokenProcedure: 1672.Reservation Segment Obj: 1070, 1071.ReadTypeList: 1804, 1807, 1809, 1917. ReservationSegmentPtr: 1070, 1194, 1257, 1260, Read Visible: 1837, 1842, 1843, 1845, 1851, 1853, 1277, 1315, 1371, 1527. 1855, 1861. Reset: 70, 625, 630, 632, 635, 640, 655, 743. ReadWord: 1682, 1683, 1688, 1689, 1690, 1692, reset: 736, 742, 756, 805, 887, 1189, 1192. 1695, 1696, 1698, 1700, 1702, 1705, 1706, ResetAccOptions: 98, 100. 1708, 1709, 1711, 1714, 1718, 1725, 1727, result: 676, 677, 678, 679, 1013, 1038, 1185, 1186, 1729, 1731, 1733, 1739, 1744, 1746, 1749, 1751, 1279, 1280, 1287, 1288, 1289, 1290, 1291, 1754, 1757, 1758, 1760, 1762, 1766, 1767, 1768, 1292, 1293, 1294, 1295, 1296, 1297, 1298, 1769, 1770, 1774, 1780, 1787, 1789, 1790, 1791, 1299, 1300, 1301, 1302, 1303, 1304, 1305, 1306, 1792, 1794, 1796, 1798, 1805, 1807, 1809, 1811, 1307, 1308, 1309, 1310, 1311, 1312, 1313, 1314, 1813, 1817, 1818, 1819, 1820, 1828, 1834, 1840, 1315, 1316, 1317, 1318, 1319, 1320, 1321, 1324, 1842, 1851, 1855, 1858, 1859, 1868, 1869, 1870, 1325, 1326, 1327, 1381. 1871, 1873, 1875, 1877, 1879, 1881, 1883, 1885, Result: 610, 1185, 1186. 1887, 1891, 1892, 1894, 1895, 1898, 1900, 1902, Rewrite: 135, 140, 152, 626, 694, 744. 1903, 1904, 1906, 1908, 1910, 1911, 1913, 1914, rewrite: 759, 1329. 1915, 1917, 1919, 1920. RightBracketName: 1187, 1189, 1190, 1191, 1228, Reasoning: 1802, 1803, 1817, 1830, 1832, 1833. 1244, 1348, 1365. ReasPos: 1802, 1830. RightCircumfixSymbol: 884, 1700, 1705, 1851. Reconsider (statement): 1079. RightSideOfPredicativeFormulaObj: 970, 971, 972. reconsider: 1079. RightSideOfPredicativeFormulaPtr: 970, 1025, record: 8. 1216, 1297, 1345, 1643. $record_separator: \underline{770}, 777, 797.$ Ritchie, Dennis M.: 1102. ReduceRegistrationObj: 1183, 1184. ROne: 257.ReduceRegistrationPtr: 1183, 1274, 1323, 1371, RoundedBracket: 882, 1191. 1455. rqRealAdd: 287.Reduction: 1904, 1910. rgRealDiff: 289.ReductionProcessing: 92, 96, 98. rqRealDiv: 297.ReferenceKind: 1047.rgRealMult: 294.ReferenceKindName: 1250, 1251, 1314. rgRealNeg: 291.ReferenceObj: 1047, 1048, 1050. rs: 226, 227, 229. ReferencePtr: 1047, 1251, 1356, 1556. RTErrorCode: 128, 146, 147, 148, 150, 155, 156. ReferenceSort: 884, 1555, 1794, 1798. Rudnicki, Piotr: 1100. Refuted: 528, 531, 536, 537, 538. RunError: 146, 147, 148, 150, 308, 336, 466, 487, RegisterCluster: 1898, 1910. 544, 562, 580, 598, 681, 685. RegisterProperty: 1908, 1910. RunTimeError: 133, 149, <u>150</u>, 160, 164, 652, 837, RegistrationBlock: 1910, 1923. 844, 845, 846, 847, 1658, 1695. RegularStatement: 1805, 1819, <u>1832</u>. RuntimeError: 838. RegularStatementKind: 1089, 1090, 1091, 1092, RZero: 257, 269. 1405. r1: 260, 261, 262, 263, 264, 265, 266, 267, 270,Regular Statement Name: 1259, 1321.271, 272, 273, 274, 275. RegularStatementObj: 1089, 1090, 1091, 1093. r2: 260, 261, 262, 263, 266, 267, 270, 271, 272,RegularStatementPtr: 1089, 1194, 1256, 1259, 273, 274, 275. 1260, 1277, 1321, 1328, 1354, 1371. 85, 281. $remaining_digits_are_zero$: 229. Save Dct: 753, 759. RenameFile: 49, 50.SaveXDct: <u>753</u>, 760, 893. $report_mismatch$: 664. $scan_right: 413, 414.$ ReportTime: 179, 180, 189, 191. scanner: <u>748</u>, 807, 879, 1030. Repr: 722, 723, 724, 725, 726, 727, 728, 730, 747. SchemeBlock: 1915, 1923. Reprs: 733, 734, 735, 737, 738, 740, 747. SchemeJustificationObj: 1061, 1062.

SchemeJustificationPtr: 1061, 1194, 1253, 1254, setlength: 603, 1011, 1025, 1029, 1189, 1192, 1388, 1277, 1319, 1328, 1358, 1359, 1360, 1549, 1570, 1584, 1601, 1609. 1557, 1559, 1560. SetLimit: 322, 325, 327, 328, 332, 333, 343, 346, SchemeListsProcessing: 92, 96, 98, 99, 100. 347, 350, 351, 357, 367, 370, 371, 373, 378, 379, 387, 388, 390, 391, 393, 394, 409, 418, 419, 486, SchemeObj: 1068, 1069. 488, 489, 490, 492, 499, 500, 507, 509, 510, 538. $SchemePos: \underline{1915}, \underline{1920}.$ SetParserPragma: 1373, 1375, 1815. SchemePragmaIntervals: 1373, 1377, 1378. SetSorted: 452, 481, 482. SchemePragmaOff: <u>1373</u>, 1376, 1377, 1378. SchemePragmaOn: 1373, 1376, 1377, 1378. SetStringLength: 36, 38.SetSym: 882, 1191. SchemePtr: 1068, 1194, 1258, 1261, 1277, 1316, settextbuf: 805.1323, 1328, 1370, 1371, 1430. SchemeSegmentKind: 1064, 1065. SetTextBuf: 655, 686.ShortString: 311. SchemeSegmentName: 1261, 1323. shr: 9.SchemeSegmentObj: 1064, 1065, 1066. SchemeSegmentPtr: 1064, 1066, 1261, 1371, 1523. SIGINT: 160.Signal Handler: 160.SchemesProcessing: 92, 96, 98. Signature Processing: <u>92</u>, 96, 98, 99, 100. $screaming_run_on_case$: 32. SIGQUIT: 160.scTooLongLineErrorNr: 882, 886, 1677. SIGTERM: 160.Search: 387, 404, 428, 432, 433, 434, 435, 447, SimpleDefiniens: 1126, 1131, 1248, 1310, 1366. 497, 499, 501, 503, 504, 510, 579, 583, 587, SimpleDefiniensObj: 1130, 1131. 589, 590, 592, 593, 594, 595, 597, 601, 605, SimpleDefiniensPtr: 1130, 1248, 1310, 1366, 1565. 607, 608, 610, 755, 784. SimpleFraenkelTermObj: 942, 943, 944. SearchPair: 506, 510, 512, 519, 520, 525, 526, 527, 531, 532, 534, 535, 537. SimpleFraenkelTermPtr: 942, 1010, 1026, 1028, Sedgewick, Robert: 399. 1029, 1234, 1301, 1348, 1619. seekEOF: 736. SimpleJustification: 1800, 1803, 1805, 1813, 1823, 1832, 1834, 1885. seekEof: 630, 632, 635, 641, 757, 1189, 1192. SegmentKind: 901, 902. SimpleJustificationObj: 1057, 1058, 1059, 1061. SimpleJustificationPtr: 1057, 1080, 1081, 1086, SegmentKindName: 1204, 1205, 1293, 1301, 1323. 1087, 1095, 1097, 1323, 1435, 1437, 1553, 1556. SelectorName: 1187, 1189, 1190, 1224, 1230, Simple Term Obj: 916, 917.1271, 1348, 1371. SimpleTermPtr: 916, 1010, 1026, 1029, 1194, SelectorSymbol: 884, 1706, 1866, 1880. 1222, 1225, 1277, 1298, 1328, 1346, 1348, 1580. SelectorTermObj: 932, 933.Single Assumption: 1150, 1153, 1272, 1323, 1362. SelectorTermPtr: 932, 1029, 1230, 1301, 1348, SingleAssumptionObj: 1152, 1153. 1627.Self: 313, 314, 315, 329, 347, 480, 514, 517, 523, SingleAssumptionPtr: 1152, 1272, 1323, 1362, 1439. 529, 539, 577, 838. Singleton: 349, 366. SemErr: 1433, 1589, 1623, 1672, 1684. $skip_all_other_ids$: 662.Semicolon: 1680, 1816, 1822, 1823, 1869, 1870, $skip_spaces$: 660. 1888, 1894, 1898, 1904, 1906, 1908, 1910, $skip_to_begin$: 1921. 1920, 1923. $skip_to_quotes: \underline{660}.$ sErrProperty: 880, 882, 1413. SetAttribute Value: 661, 673. $skip_xml_prolog:$ 662. $skipped_proof_justification: 1399, 1400.$ SetCapacity: 452, 454, 455, 462, 469, 480, 552, Skyscraper: 1100. 554, 557, 558, 560, 565, 567, 569, 579, 581, 582, 583, 585, 597, 599, 600, 601, 603. SliceIt: <u>764</u>, 773, 789. SetConsoleCtrlHandler: 162, 163. Soft type: 956. SetConsoleMode: 163.SOLARIS: 87. SethoodRegistrationObj: 1181, 1182.Sort: 387, 400, 407, 452, 481, 482. SethoodRegistrationPtr: 1181, 1274, 1323, 1371, Source: 70. 1455, 1492. Specification: <u>1856</u>, 1866, 1917.

Spelling: 764, 765, 766, 767, 881, 886, 889, 1395, 1815. SplitFileName: 53, 54, 56, 58, 60, 62. SquareBracket: 882, 1191, 1501. 1747. StackedObj: 449, 450, 853, 863, 873. StackedPtr: 449.StandardModeDefinitionObj: 1140, 1141. StandardModeDefinitionPtr: 1140, 1268, 1323, 1371, 1444. StandardPriority: 715, 724, 727. Standard Type Obj: 960, 961. Standard TypePtr: 960, 1021, 1201, 1290, 1342, 1589. Start: 1567, 1570, 1607, 1609, 1610, 1639. StartAdjectiveCluster: 877, 878, 1724. StartAggregateTerm: 876, 878, 1624, 1698. StartAggrPattSegment: <u>866</u>, 868, 1512, 1866. 1809. StartArgument: <u>877</u>, 878, 1737. StartAssumption: 866, 868, 1474, 1787. StartAtomicFormula: 877, 878, 1760. StartAtSignProof: 853, 857, 1802. StartAttributeArguments: 876, 878, 1576, 1725, 1897. StartAttributePattern: <u>866</u>, 868, 1490, 1845. StartAttributes: 866, 868, 876, 878, 1459, 1571, 1735, 1758, 1895, 1898, 1900, 1902, 1903. 1832. StartAttrIdentify: 867, 868. StartBracketedTerm: <u>876</u>, 878, 1622, 1690, 1702. StartChoiceTerm: 876, 878, 1630, 1706, 1709, 1711. StartCollectiveAssumption: 866, 868, 1476, 1780. StartCondition: <u>867</u>, 868, 1785. StartConstructionType: 866, 868, 1544, 1868. StartDefiniens: 866, 868, 1481, 1873. StartDefPredicate: 867, 868, 1870, 1881, 1883. 1798. StartEquals: 867, 868, 1877. StartExemplifyingTerm: 866, 868, 1542, 1818. StartExistential: 876, 878, 1661, 1742. StartExpansion: 866, 868, 1487, 1871. StartFields: 866, 868, 1509, 1858. StartFixedSegment: 866, 868, 1468, 1783. StartFixedVariables: 866, 868, 1466, 1782. StartForgetfulTerm: <u>876</u>, 878, 1628, 1709. StartFraenkelTerm: 876, 878, 1611, 1703, 1711. StartFuncIdentify: 866, 868, 1463, 1906. StartFuncReduction: 866, 868, 1465, 1904. StartFunctorPattern: 866, 868, 1501, 1847. StartGuard: 866, 868, 1483, 1875, 1879. $StartIterativeStep \colon \quad \underline{866}, \ 868, \ \underline{1563}, \ 1834.$ StartJustification: 866, 868, 1558, 1803, 1832, 1833, 1834.

StartLibraryReferences: 866, 868, 1551, 1794.

StartLongTerm: 876, 878, 1593, 1714. StartModePattern: <u>866</u>, 868, 1493, 1838. StartMultiPredicativeFormula: 876, 878, 1644, StartNewType: <u>867</u>, 868, 1813. StartOtherwise: 867, 868, 1875, 1879. StartPostqualification: 876, 878, 1612, 1691. StartPostqualificationSpecyfication: 876, 878,1615, 1692. StartPostqualifyingSegment: 876, 878.StartPostQualifyingSegment: 1613, 1692. StartPredicatePattern: <u>866</u>, 868, 1495, 1853. StartPredIdentify: 867, 868. StartPrefix: 867, 868, 1859. StartPrivateConstant: <u>866</u>, 868, 1533, 1811. StartPrivateDefiniendum: 866, 868, 1529, 1807, StartPrivateDefiniens: 866, 868, 1534, 1807, 1809. StartPrivateFormula: <u>876</u>, 878, 1648, 1762. StartPrivateTerm: 876, 878, 1620, 1696. StartProperText: 853, 857, 1385, 1396, 1922. StartQualifiedSegment: 876, 878, 1663, 1739. StartQualifyingType: 876, 878, 1664, 1739. StartReferences: <u>867</u>, 868, 1791. StartRegularStatement: 866, 868, 1546, 1777, StartReservationSegment: 866, 868, 1527, 1911. StartRestriction: 876, 878, 1656, 1744. $starts_with_term_token$: $\underline{1760}$. StartScaner: 883, 889, 890, 892. StartScanner: 886.StartSchemeDemonstration: 853, 860, 1385, 1399, 1920. StartSchemeLibraryReference: 866, 868, 1552, StartSchemePremise: 867, 868, 1919. StartSchemeQualification: 866, 868, 1517, 1917. StartSchemeReference: 867, 868, 1796. StartSchemeSegment: <u>866</u>, 868, 1521, 1917. StartSelectorTerm: <u>876</u>, 878, 1626, 1708. StartSentence: 866, 868, 1460, 1775, 1776. StartSethoodProperties: 867, 868, 1908. StartSimpleFraenkelTerm: <u>876</u>, 878, 1618, 1711. StartSimpleJustification: 866, 868, 1559, 1800. StartSpecification: 867, 868, 1856. StartTheoremBody: 867, 868, 1913, 1914. StartType: 876, 878, 1587, 1735, 1758, 1898, 1900, 1902, 1903. StartUniversal: 876, 878, 1662, 1744. Start Visible: 866, 868, 1504, 1837, 1843. State variable: 36. StAX, Java: 688.

stCompactStatement: 1089, 1094, 1256, 1259, Str2PropertyKind: 1286, 1323. 1321, 1354, 1441, 1546. Str2TermKind: 1286, 1301. STD_INPUT_HANDLE: 163. Str2XMLAttrKind: 1284. stdcall: 165. Str2XMLElemKind: 1283.stDiffuseStatement: 1089, 1092, 1256, 1259, 1321, Sub: 215, 229, 233, 241, 242, 243, 244, 246, 1354, 1441, 1546. 247, 263. $steal_from: 398.$ $SubexpObj\colon \ 873,\ \underline{874},\ \underline{875},\ \underline{878},\ 1568.$ StillCorrect: 1456, 1494, 1499, 1503, 1560, 1623, SubexpPtr: 850, 872, 873, 875. 1629, <u>1672</u>, 1677, 1682, 1684, 1816, 1825, $swap_indices$: 398. 1828, 1829, 1869, 1870, 1885, 1888, 1893, SwitchOffUnifier: 93, 105. 1910, 1920, 1923. sx: 216.still correct: 1491. $sy_according:$ 884. stIterativeEquality: 1089, 1098, 1256, 1259, 1321, $sy_aggregate$: 884. 1354, 1441, 1561. $sy_all: 884.$ StopOnError: 93, 99, 100, 103, 105, 130, 131, 191. sy_All: 1348, 1711. Store: 722, 729, 740. sy_Ampersand: 884, 1343, 1658, 1766, 1767. StoreFIL: 622, 626. $sy_{-}and: 884.$ StoreFormats: 825, 839. sy_And: 1361, 1371, 1778, 1785, 1919. StoreMmlVcb: 717, 744. $sy_antonym$: 884. StoreMmlVcbX: 717, 745. $sy_Antonym: 1371, 1894.$ Store Voc: <u>733</u>, 740, 744. $sy_are: 884, 1692.$ Str: 46, 85, 86, 178, 180, 182, 213, 214, 216, 218, sy_Arrow: 884, 1371, 1544, 1856, 1868, 1898, 219, 220, 224, 281, 642, 727, 765. 1900, 1904. StraightforwardJustificationObj: 1059, 1060. *sy_As*: 1371, 1813. StraightforwardJustificationPtr: 1059, 1252, 1254, *sy_as*: 884. 1277, 1318, 1328, 1354, 1357, 1360, 1371, 1559. sy_Assume: 1371, 1817, 1891. StreamObj: 680, 681, 682, 683, 693. sy_assume : 884. StrictSym: 882, 1191. sy_Attr: 1371, 1870, 1888, 1889. String: 45, 48, 50, 52, 75, 79, 178, 179, 180, $sy_-attr:$ 884. 192, 211, 254, 259, 311. $sy_axiom:$ 884. StringColl: 437, 440, 631, 632, 637, 638. sy_Axiom : 1923. StringListError: 453, 459, 463, 466, 472, 477. sy_Be: 1339, 1739, 1783. StrToInt: 678. $sy_be: 884.$ StructModeMaxArgs: 883, 887, 888, 1731, 1863. $sy_begin: 884.$ StructTypeObj: 962, 963.sy_Begin: 886, 1371, 1678, 1802, 1817, 1827, 1888, StructTypePtr: 962, 1021, 1201, 1290, 1342, 1589. 1893, 1910, 1921, 1922, 1923, 1924. Structure, first-class: 1100. $sy_being:$ 884. StructureDefinitionObj: 1110, 1111. $sy_Being: 1739, 1783.$ StructureDefinitionPtr: 1110, 1271, 1323, 1371, *sy_By*: 1357, 1559, 1800. 1450. *sy_by*: 884. Structure Mode Max Args: 887. $sy_canceled:$ 884. StructureName: 1187, 1189, 1190, 1201, 1229, sy_Canceled: 1891, 1910, 1923. 1231, 1271, 1342, 1348, 1371. sy_Case: 1367, 1678, 1802, 1817, 1821, 1827, 1828, StructureSymbol: 884, 1507, 1589, 1698, 1706, 1714, 1716, 1720, 1725, 1727, 1758, 1760, 1805, 1885, 1891, 1910, 1923. sy_case : 884. 1861, 1880, 1895, 1897. sy_cases : 884. Structuresymbol: 1687. sy_Cases: 1371, 1823, 1885. Str2BlockKind: 1285, 1325. $sy_cluster$: 884. Str2CorrectnessKind: 1286, 1322, 1323. Str2FormulaKind: 1285, 1295. $sy_Cluster$: 1371, 1910. Str2ItemKind: 1285, 1326. sy_Colon: 884, 1545, 1548, 1702, 1703, 1774, 1794, 1798, 1873, 1877, 1915. Str2PatterenKind: 1285, 1309.

sy_Comma: 884, 1542, 1691, 1693, 1718, 1737, 1739, 1740, 1754, 1782, 1783, 1791, 1794, 1804, 1811, 1813, 1818, 1837, 1859, 1864, 1866, 1875, 1879, 1880, 1906, 1911, 1917. $sy_consider$: 884. sy_Consider: 1371, 1678, 1805. sy_Contradiction: 1345, 1760, 1805. $sy_contradiction$: 884. sy_Correctness: 1371, 1543, 1869. $sy_correctness$: 884. $sy_CorrectnessCondition$: 884, 1869. $sy_deffunc:$ 884. $sy_Deffunc: 1678, 1805.$ $sy_DefFunc:$ 1371. $sy_define:$ 884. $sy_-definition:$ 884. sy_Definition: 1367, 1678, 1802, 1817, 1827, 1888, 1893, 1910, 1923. $sy_defpred:$ 884. sy_Defpred: 1678, 1805. $sy_DefPred: 1371.$ sy_Do: 1747, 1752, 1756, 1757, 1760, 1805. $sy_{-}do: 884.$ sy_Does: 1345, 1747, 1752, 1756, 1757, 1760, 1805. sy_does : 884. sy_Dolar: 884, 1687, 1695, 1706, 1714, 1716, 1720, 1760, 1805. $sy_DotEquals$: 884, 1354, 1560, 1561, 1800, 1832. sy_Ellipsis: 884, 1343, 1766, 1767, 1768, 1769. $sy_end:$ 884. sy_End: 1367, 1678, 1680, 1802, 1816, 1821, 1827, 1887, 1892, 1910, 1923, 1925. $sy_environ:$ 884. sy_Equal: 884, 1640, 1641, 1747, 1748, 1752, 1756, 1757, 1760, 1805, 1807, 1811, 1813, 1818, 1853, 1855, 1880, 1906. sy_Equals: 1371, 1872. $sy_equals:$ 884. sy_Error: 884, 886, 1677, 1880. sy_Ex: 1345, 1744, 1760, 1805. $sy_ex: 884.$ $sy_exactly$: 884. sy_Exactly: 1348, 1689, 1713. $sy_- for:$ 884. sy_For: 1345, 1371, 1744, 1760, 1805, 1881, 1883, 1898, 1900, 1902, 1911. sy_From: 1359, 1559, 1800. $sy_from:$ 884. sy_Func: 1371, 1870, 1877, 1888, 1889. $sy_func:$ 884. $sy_qiven:$ 884. sy_Given: 1371, 1817, 1891.

 sy_hence : 884. sy_Hence: 1398, 1819. sy_Hereby: 1678, 1802, 1817, 1830, 1885, 1891, 1910, 1923. sy_hereby : 884. sy_holds : 884. sy_Holds: 1345, 1744. $sy_identify$: 884. $sy_Identify: 1371, 1910.$ sy_If: 1366, 1873, 1875, 1877, 1879. $sy_if: 884.$ $sy_{-}iff: 884.$ $sy_{-}Iff: 1343, 1658, 1770.$ $sy_implies: 884.$ sy_Implies: 1343, 1658, 1770. sy_Is: 1345, 1348, 1365, 1371, 1639, 1692, 1752, 1845, 1871, 1880. $sy_is: 884.$ sy_It: 1348, 1687, 1695, 1706, 1714, 1716, 1720, 1760, 1805. $sy_it: 884.$ sy_LeftCurlyBracket: 884, 1501, 1687, 1702, 1706, 1714, 1716, 1720, 1760, 1805, 1847, 1880, 1917. sy_LeftParanthesis: 884, 1523, 1687, 1688, 1696, 1706, 1714, 1725, 1733, 1758, 1796, 1805, 1807, 1843, 1845, 1847, 1859, 1880, 1895, 1917. sy_LeftSquareBracket: 884, 1501, 1523, 1687, 1700, 1706, 1714, 1716, 1720, 1760, 1805, 1809, 1847, 1880, 1917. $sy_let:$ 884. sy_Let: 1371, 1817, 1891, 1894, 1910. sy_LibraryDirective: 884. $sy_means:$ 884. sy_Means: 1371, 1809, 1872. sy_Mode: 1371, 1870, 1888, 1889. sy_mode : 884. sy_non : 884. sy_Non: 1335, 1572, 1706, 1725, 1758, 1895. sy_not : 884. sy_Not: 1345, 1749, 1757, 1758, 1760, 1805. $sy_notation$: 884. sy_Notation: 1367, 1678, 1802, 1817, 1827, 1888, 1893, 1910, 1923. $sy_now:$ 884. sy_Now: 1367, 1546, 1678, 1802, 1805, 1830, 1832. $sy_{-}Of$: 1342, 1348, 1365, 1371, 1588, 1627, 1708, 1709, 1711, 1729, 1840, 1842. $sy_{-}of:$ 884, 1908. sy_Or: 1343, 1658, 1768, 1769. *sy_or*: 884. sy_Otherwise: 1366, 1875, 1879. $sy_otherwise:$ 884.

sy_Over: 1342, 1371, 1731, 1861. sy_Such: 1371, 1473, 1785, 1787. $sy_over:$ 884. sy_such : 884. sy_per : 884. $sy_suppose$: 884. sy_Per: 1371, 1678, 1817, 1820, 1821, 1822, 1823, sy_Suppose: 1367, 1678, 1802, 1817, 1821, 1825, 1885, 1891, 1910, 1920, 1923. 1827, 1885, 1891, 1910, 1923. sy_Pred: 1371, 1870, 1888, 1889. $sy_synonym$: 884. sy_Synonym: 1371, 1894. $sy_pred:$ 884. $sy_prefix:$ 884. sy_Take: 1371, 1817. $sy_take:$ 884. sy_proof : 884. sy_Proof: 1367, 1371, 1558, 1678, 1802, 1803, sy_That: 1361, 1371, 1417, 1419, 1779, 1785, 1787, 1885. 1805, 1830, 1832, 1920. sy_Property: 884, 1870, 1910. $sy_that:$ 884. $sy_provided$: 884. $sy_{-}the:$ 884. $sy_Provided$: 1371, 1919. sy_The: 1348, 1687, 1706, 1714, 1716, 1720, *sy_qua*: 884. 1760, 1805, 1880. sy_Then: 1353, 1398, 1678, 1805, 1817, 1819. sy_Qua: 1348, 1689, 1713. $sy_reconsider$: 884. $sy_then:$ 884. $sy_theorem: 884.$ sy_Reconsider: 1371, 1678, 1805. $sy_redefine:$ 884. sy_Theorem: 1371, 1678, 1802, 1817, 1827, 1888, $sy_Redefine: 1371, 1397, 1888.$ 1893, 1910, 1923. sy_Thesis: 1345, 1760, 1805. sy_Reduce: 1371, 1910. $sy_thesis:$ 884. sy_reduce : 884. $sy_registration$: 884. *sy_Thus*: 1371, 1819. $sy_Registration$: 1367, 1678, 1802, 1817, 1827, $sy_thus:$ 884. sy_To: 1371, 1904. 1888, 1893, 1910, 1923. *sy_to*: 884. *sy_reserve*: 884, 1371. sy_When: 1371, 1463, 1906. sy_Reserve: 1678, 1802, 1817, 1827, 1888, 1893, 1910, 1923. $sy_when: 884.$ sy_RightCurlyBracket: 884, 1700, 1703, 1705, $sy_where:$ 884. sy_Where: 1348, 1691, 1702. 1851, 1915. sy_RightParanthesis: 884, 1688, 1696, 1700, 1705, sy_With: 1371, 1906. 1714, 1718, 1719, 1722, 1734, 1754, 1764, 1796, $sy_with:$ 884. 1804, 1807, 1843, 1845, 1851, 1859, 1917. $sy_wrt:$ 884. sy_RightSquareBracket: 884, 1700, 1705, 1762, syAbstractness: 880. syAssociativity: 880, 1413. 1804, 1809, 1851, 1917. sy_Scheme: 1371, 1426, 1678, 1802, 1817, 1827, syAsymmetry: 880, 1413. 1888, 1893, 1910, 1923. syCoherence: 880, 1424, 1452, 1544, 1565. syCommutativity: 880, 1413. sy_scheme : 884. $sy_selector:$ 884. syCompatibility: 880, 1429, 1455. syConnectedness: 880, 1413. sy_Semicolon: 884, 1542, 1560, 1678, 1680, syConsistency: 880, 1429. 1800, 1925. syConstructors: 880. sy_set : 884. sy_Set: 1348, 1371, 1588, 1706, 1725, 1727, 1758, syCorrectness: 880, 1259. 1805, 1838, 1895. syDef: 880, 1555, 1794. sy_St: 1345, 1742, 1744. syDefinitions: 880. *sy_st*: 884. syEqualities: 880. sy_Struct: 1371, 1870, 1888. syExistence: 880, 1444, 1452, 1565. sy_struct : 884. syExpansions: 880. sy_StructLeftBracket: 884, 1348, 1371, 1698, 1725, syIdempotence: 880, 1413. sy Involutiveness: 880, 1413. 1864, 1895, 1897. sy_StructRightBracket: 884, 1348, 1371, 1698, syIrreflexivity: 880, 1413. SymbolCounters: 714, 731. 1864.

 $\S1926$ Mizar Parser INDEX 631

SymbolIntSeqArr: 714.	syT129: 884.
$SymbolStr: \underline{722}, 727, 729.$	syT13: 884.
symPri: 1607, 1608.	syT130: 884.
SymPri: 1567, 1597, 1607.	syT131: 884.
SynErr: <u>1677</u> , 1679, 1725, 1748, 1757, 1895,	syT132: 884.
1898, 1908.	syT133: 884.
Synonym: <u>1881</u> , 1894.	syT134: 884.
syNotations: 880.	syT136: 884.
syntax: 841, 895, 1030, 1379, 1672.	syT137: 884.
syntax.txt: 1035.	syT138: 884.
syProjectivity: 880, 1413.	syT139: 884.
syReducibility: 880, 1455.	syT14: 884.
syReflexivity: 880, 1413.	syT141: 884.
syRegistrations: 880.	syT142: 884.
syRequirements: 880.	syT143: 884.
sySch: 880, 1798.	syT146: 884.
sySchemes: 880.	syT148: 884.
sySethood: 880, 1274, 1323, 1371, 1413, 1908.	syT149: 884.
System T: 1102.	syT15: 884.
$\overset{\circ}{SystemTime}$: 173, 175.	syT150: 884.
SystemTimeToMiliSec: 171, 173, 175.	syT151: 884.
SysUtils: 37, 47, 48, 50, 54, 64, 674.	syT152: 884.
sySymmetry: 880, 1413.	syT153: 884.
syThe: 880.	syT154: 884.
syTheorems: 880.	syT155: 884.
syTransitivity: 880, 1413.	syT158: 884.
syT0: 884.	syT16: 884.
syT1: 884.	syT160: 884.
syT10: 884.	syT163: 884.
syT100: 884.	syT17: 884.
syT102: 884.	syT173: 884.
syT103: 884.	syT177: 884.
syT105: 884.	syT179: 884.
syT106: 884.	syT18: 884.
syT107: 884.	syT180: 884.
syT108: 884.	syT181: 884.
syT109: 884.	syT182: 884.
syT11: 884.	syT183: 884.
syT110: 884.	syT184: 884.
syT111: 884.	syT186: 884.
syT112: 884.	syT187: 884.
syT113: 884.	syT189: 884.
syT115: 884.	syT19: 884.
syT117: 884.	syT190: 884.
syT118: 884.	syT191: 884.
syT12: 884.	syT194: 884.
syT121: 884.	syT195: 884.
syT122: 884.	syT198: 884.
syT124: 884.	syT199: 884.
syT126: 884.	syT2: 884.
syT127: 884.	syT20: 884.
syT128: 884.	syT207: 884.
U	

TTOO O OO I	TTO 0 004
syT208: 884.	syT96: 884.
syT209: 884.	syT98: 884.
syT21: 884.	syUniqueness: 880, 1565.
syT210: 884.	sy Vocabularies: 880.
syT216: 884.	TByteArray: 311.
syT217: 884.	TCharSet: 311.
syT218: 884.	$\overline{TElementState}$: 648, 661.
syT219: 884.	Temp: 464.
syT22: 884.	$\overline{TempObject}$: $\underline{476}$.
syT220: 884.	TempString: 476.
syT221: 884.	Term: 1388, <u>1389</u> , 1573, 1582, 1583, 1584, 1585,
syT222: 884.	1586, 1646, 1647.
syT223: 884.	Term rewriting: 1183.
syT23: 884.	Term, Bracket: 926.
syT24: 884.	Term, Fraenkel: 944.
syT25: 884.	
· ·	TermBegSys: 1687, 1725, 1746, 1751, 1758,
·	1895, 1897.
syT27: 884.	TermExpression: <u>1773</u> , 1807, 1811, 1813, 1818,
syT28: 884.	1834, 1877, 1879, 1904.
syT29: 884.	TermExpressionObj: 913, 916, 918, 920, 922, 924,
syT3: 884.	930, 934, 940, 942, 948, 950, 952, 954.
syT30: 884.	TermKindLookupTable: 1281, 1282, 1286.
syT31: 884.	TermName: 1222, 1223, 1224, 1225, 1226, 1227,
syT4: 884.	1228, 1229, 1230, 1231, 1232, 1233, 1234,
syT48: 884.	1235, 1236, 1237, 1281.
syT49: 884.	TermNbr: 28, <u>1569</u> , 1570, 1573, 1575, 1576, 1577,
syT5: 884.	1579, 1582, 1583, 1584, 1585, 1586, 1589, 1593,
syT50: 884.	1609, 1610, 1623, 1625, 1636, 1639, 1640, 1641,
syT51: 884.	1642, 1643, 1646, 1647, 1648, 1670.
syT52: 884.	TermPtr: 913, 930, 931, 932, 933, 938, 939, 942,
syT53: 884.	943, 944, 945, 946, 947, 948, 949, 976, 977,
syT54: 884.	980, 981, 1010, 1020, 1029, 1073, 1074, 1077,
syT55: 884.	1078, 1080, 1081, 1083, 1084, 1095, 1097, 1173,
syT56: 884.	1174, 1183, 1184, 1194, 1225, 1248, 1277, 1295,
syT57: 884.	1301, 1321, 1323, 1328, 1348, 1366, 1389, 1407,
syT6: 884.	1451, 1453, 1568, 1606.
syT64: 884.	TermSort: 913, 924, 925, 930, 931, 1281, 1286.
syT66: 884.	TermSubexpression: 1691, 1702, 1708, 1709, 1711,
syT67: 884.	<u>1720,</u> 1737, 1738, 1773.
syT69: 884.	TermWithArgumentsObj: 924, 925, 926, 928, 936.
syT7: 884.	TermWithArgumentsPtr: 924.
syT70: 884.	text: 133, 151, 625, 626, 630, 632, 634, 638, 639,
syT72: 884.	653, 684, 717, 722, 728, 729, 733, 736, 738, 740,
syT8: 884.	742, 743, 744, 756, 759, 792, 887, 1188.
syT80: 884.	The Art of Computer Programming: 399.
syT83: 884.	the (Choice operator): 950.
syT84: 884.	the_item: 778, 780.
syT87: 884.	the_token: 778, 779, 780.
syT9: 884.	Theorem: 1913, 1923.
· ·	TheoremListsProcessing: 92, 96, 98, 99, 100.
·	
syT94: 884.	Theorem Reference: 1047, 1052, 1251, 1314, 1356.
syT95: 884.	TheoremReferenceObj: 1051 , $\underline{1052}$.

TheoremReferencePtr: 1051, 1251, 1314, 1356, TypeChangeObj: 1080, 1081.TypeChangePtr: 1080, 1263, 1323, 1371, 1537, 1556. TheoremsProcessing: 92, 96, 98. 1538. TypeChangeSort: 1080, 1081. thesis: 1006. $thesis_formula: 1399, 1638.$ TypeChangingStatementObj: 1080, 1081. $thesis_prop: 1399, 1400.$ TypeChangingStatementPtr: 1080, 1263, 1323, 1371, 1435. ThesisFormulaObj: 1006, $\underline{1007}$. ThesisFormulaPtr: 1006, 1295, 1399. TypeExpression: <u>1773</u>, 1783, 1804, 1813, 1856, 1859, 1868, 1871, 1908. Time: 619, 620, 624, 626. TimeMark: 172, 173, 184, 187. TypeExpressionObj: 957, 958, 964, 966. TypeName: 1201, 1290. TIntItem: 441, 442, 443. $TIntKeyCollection: \underline{441}, \underline{443}, \underline{444}.$ TypePtr: 905, 906, 946, 947, 950, 951, 957, 964, To_Right: 1597, 1601, 1602, 1607, 1608. 965, 980, 981, 1010, 1021, 1066, 1067, 1070, Token: 881, 883. 1071, 1080, 1081, 1106, 1107, 1138, 1139, 1140, 1141, 1146, 1147, 1167, 1168, 1169, 1170, 1171, Tokenise, reserved keywords: 779. 1172, 1173, 1174, 1181, 1182, 1194, 1201, 1277, Tokenization: 886. 1290, 1323, 1328, 1342, 1369, 1407, 1408. TokenKind: 881, 884, 885, 886, 889, 1500, 1568, TypeSort: 957, 958, 959. 1678, 1683, 1687, 1714, 1821, 1870, 1880. TokenName: 885, 1335, 1339, 1342, 1343, 1345, TypeSubexpression: 1689, 1692, 1706, 1709, 1711, 1348, 1353, 1354, 1357, 1359, 1361, 1365, 1735, 1739, 1773, 1911. 1366, 1367, 1371. UncapitalizeName: 1186.Unexpected: 882. TokenObj: 751, 752, 761.UnexpectedItem: <u>1885</u>, 1891, 1894, 1910, 1923. TokenPtr: 751, 755, 757, 758, 759, 760, 766, 767, 772, 778, 783, 784. UnexpectedXMLElem: 646, 652. unit: $\frac{7}{2}$. Tokens Collection: 753, 754, 755, 756, 759, 760, 764. unit_separator: 770, 776, 798. TokensSet: 648, 661, 664. UniversalFormula: 1744, 1760. $\label{linear_equation} Universal Formula Obj: \quad 1000, \ \underline{1001}.$ Total functional programming: 1102. TPath: 54.Universal Formula Ptr: 1000, 1295, 1668.TransferItems: 322, 347, 364. Unused code: 629, 631. Up: 528, 532, 579, 592.*Trim*: 677. trimlz: 200, 203, 211, 215, 219, 221, 224, 227, UpCase: 44, 107.229, 233. $update_lexeme: 657.$ TrimString: 41, 42, 635, 718, 720, 721, 724, 725, $UpperCase: 43, \underline{44}, 75.$ 726, 728, 737, 1374. uses: $\frac{7}{}$. TrimStringLeft: 39, 40, 42. UTF-8: 45, 657, 748. TrimStringRight: 39, 40, 42.Val: 1280, 1374. tring: 74.val: 207, 213, 214, 216, 218, 220, 223, 639, TruncDir: 55, 56.692, 720, 726. TruncExt: 57, $\underline{58}$. ValidCharTable: 676. $Try_read_ini_var$: 639, 641. Value: 528, 531, 535, 541, 542, 543, 547, 550, 552, Trybulec, Andrzej: 15, 807. 563, 579, 591, 594, 597, 610, 1602, 1608. $TSymbol: 28, 722, \underline{723}, \underline{724}, \underline{727}, \underline{728}, \underline{729}, \underline{730}.$ var: 7. TSystemTime: 170, 171, 173, 175. Variable, Environment: 65, 617. Turing complete: 1102. Variable, global: 36. Turner, David A.: 1102. Variable, state: 36. TVocabulary: 733, 734, 735, 736, 738, 740.VariableIdentifier: 1080, 1263, 1323, 1371, 1538. TWordArray: 311.VariableObj: 898, 899.TXTStreamObj: 28, 684, 685, 686, 687, 1328. VariablePtr: 898, 903, 904, 1010, 1016, 1018, Type, radix: 1727. 1073, 1074, 1075, 1076, 1077, 1078, 1080, Type, Soft: 956. 1081, 1083, 1084, 1194, 1202, 1203, 1277, Type, Standard: 960. 1291, 1293, 1323, 1328, 1337, 1469, 1471, 1522,

1527, 1532, 1535, 1536, 1537, 1538, 1541, 1542, 1614, 1616, 1665, 1666. VCXfile: 745, 746, 747. VERALPHA: 86. VerifyPragmaIntervals: 1373, 1377, 1378. VerifyPragmaOff: 1373, 1376, 1377, 1378. VerifyPragmaOn: <u>1373</u>, 1376, 1377, 1378. VersionStr: 84, 86, 88. VER70: 313. ViableFormula: 1760, 1766, 1767, 1768, 1769, 1770, 1772. VocabulariesProcessing: 92, 96, 98, 99, 100. W: 172, 173.WeakerThan: $\underline{528}$, 541, 543. Weakly strict Mizar: 1030. wHour: 170, 171. Wiedijk, Freek: 15, 956. WINBOOL: 165.windows: 37, 154, 169, 617. WIN32: 83, 87, 113, 154, 160, 161, 617. Wirth, Niklaus: 7, 33. WithinExprObj: 1010, 1011, 1012, 1013, 1014, 1015, 1016, 1017, 1018, 1019, 1020, 1021, 1022, <u>1023</u>, <u>1024</u>, <u>1025</u>, <u>1026</u>, <u>1027</u>, <u>1028</u>, <u>1029</u>. Within ExprPtr: 1010.without_io_checking: <u>13</u>, 70, 135, 140, 736. wMilliseconds: 170, 171.wMinute: 170, 171.word: 145, 146, 147, 148, 149, 150, 155, 170, 176, 177, 178, 180, 311, 313, 714, 792. Write: 740. write: 109, 111, 113, 116, 117, 119, 151, 689, 694, 1330, 1332, 1333. $Write_WSMizArticle: 1276.$ WriteError: 132, 133, 134, 135, 191, 193. WriteLn: 109, 111, 113, 114, 116, 119, 126, 133, 151, 152, 198, 202, 203, 206, 208, 209, 210, 212, 215, 219, 222, 224, 226, 230, 233, 235, 237, 239, 244, 249, 251, 626, 689, 694, 729, 740. writeln: 759, 1331. Wrong Word: 1679, 1706, 1716, 1721, 1729, 1731, 1758, 1760, 1770, 1792, 1796, 1800, 1805, 1830, 1838, 1840, 1847, 1853, 1877, 1920. wsAdjective: 907, 912, 1014, 1199, 1288, 1335. wsAggregateTerm: 913, 937, 1029, 1225, 1229, 1301, 1348. wsAttributiveFormula: 968, 977, 1025, 1207, 1218, 1295, 1345. wsBiconditionalFormula: 968, 993, 1025, 1207, 1212, 1296, 1343, 1345. wsBlock: 1036, 1039, 1387.

WSBlockPtr: 1038, 1194, 1197, 1325, 1328, 1367, 1382, 1385. wsBlockPtr: 1036, 1038, 1040, 1194, 1254, 1255, 1256, 1277, 1325, 1328, 1352, 1354, 1360. wsCircumfixTerm: 913, 927, 1029, 1225, 1228, 1301, 1348. wsClusteredType: 957, 965, 1021, 1201, 1290, 1342. wsConditionalFormula: 968, 991, 1025, 1207, 1211, 1296, 1343, 1345. wsConjunctiveFormula: 968, 987, 1025, 1207, 1209, 1296, 1343, 1345. wsContradiction: 968, 1005, 1025, 1207, 1295, wsDisjunctiveFormula: 968, 989, 1025, 1207, 1210, 1296, 1343, 1345. wSecond: 170, 171.wsErrorFormula: 968, 1009, 1025, 1207, 1285, 1295, 1345. wsErrorTerm: 913, 955, 1029, 1225, 1286, 1301, 1348. wsErrorType: 957, 967, 1021, 1201, 1342. wsExactlyTerm: 913, 949, 1029, 1225, 1301, 1348. wsExistentialFormula: 968, 1003, 1025, 1207, 1221, 1295, 1345. wsFlexaryConjunctiveFormula: 968, 995, 1025, 1207, 1213, 1295, 1343, 1345. wsFlexaryDisjunctiveFormula: 968, 997, 1025, 1207, 1214, 1295, 1343, 1345. wsForgetfulFunctorTerm: 913, 939, 1029, 1225, 1231, 1301, 1348. wsFraenkelTerm: 913, 945, 1029, 1225, 1233, 1301. 1348. wsGlobalChoiceTerm: 913, 951, 1029, 1225, 1237,1301, 1348. wsInfixTerm: 913, 923, 1029, 1225, 1227, 1301, 1348. wsInternalForgetfulFunctorTerm: 913, 941, 1029, 1225, 1232, 1301, 1348. wsInternalSelectorTerm: 913, 935, 1029, 1224, 1225, 1301, 1348. wsItem: 1040, 1041, 1387. WSItemPtr: 1194, 1259, 1260, 1275, 1326, 1328, 1371, 1382, 1385, 1405. wsItemPtr: 1036, 1038, 1040, 1277, 1322, 1323, 1326, 1391. wsItTerm: 913, 953, 1029, 1225, 1301, 1348. wsmarticle: 1030, 1379. WSMizarPrinterObj: 1328, 1329, 1330, 1331, 1332, <u>1333</u>, <u>1334</u>, <u>1335</u>, <u>1336</u>, <u>1337</u>, <u>1338</u>, <u>1339</u>, 1340, 1341, 1342, 1343, 1344, 1345, 1346, 1347, 1348, 1349, 1350, 1351, 1352, 1353, 1354, 1355,

1356, 1357, 1358, 1359, 1360, 1361, 1362, 1363,

<u>1364</u>, <u>1365</u>, <u>1366</u>, <u>1367</u>, <u>1368</u>, <u>1369</u>, <u>1370</u>, <u>1371</u>. WSMizarPrinterPtr: 1328, 1372.wsMultiPredicativeFormula: 968, 975, 1025, 1207, 1217, 1297, 1345. wsNegatedAdjective: 907, 910, 1014, 1199, 1335. wsNegatedFormula: 968, 983, 1025, 1207, 1208, 1295, 1345, 1643. $ws Numeral Term \colon \ 913, 921, 1029, 1225, 1301, 1348.$ wsPlaceholderTerm: 913, 919, 1029, 1225, 1226, 1301, 1348. wsPredicativeFormula: 968, 973, 1025, 1207, 1215, 1297, 1345. wsPrivateFunctorTerm: 913, 929, 1029, 1223, 1225, 1301, 1348. wsPrivatePredicateFormula: 968, 979, 1025, 1206, 1207, 1297, 1345. wsQualificationTerm: 913, 947, 1029, 1225, 1235, 1236, 1301, 1348. wsQualifyingFormula: 968, 981, 1025, 1207, 1219, 1295, 1345. wsReservedDscrType: 957.wsRightSideOfPredicativeFormula: 968, 971, 1025, 1207, 1216, 1297. wsSelectorTerm: 913, 933, 1029, 1225, 1230, 1301, 1348. wsSimpleFraenkelTerm: 913, 943, 1029, 1225, 1234, 1301, 1348. wsSimpleTerm: 913, 917, 1029, 1222, 1225, 1301, 1348. wsStandardType: 957, 961, 1021, 1201, 1290, 1342. wsStructureType: 957, 963, 1021, 1201, 1290, 1342. wsTextProper: 1036, 1037, 1038. wsTextProperPtr: 1036, 1276, 1277, 1324, 1327, 1372, 1382, 1383. WSTextProperPtr: 1194, 1196, 1328, 1368. wsThesis: 968, 1007, 1025, 1207, 1295, 1345. wsUniversalFormula: 968, 1001, 1025, 1207, 1220, 1295, 1345. .wsx file: 1030. $x: \ \underline{215}.$ *xml_dict*: <u>612</u>, 674, 714, 748, 807, 1030. xml_inout : 674, 714, 748, 807, 1030, 1379. *xml_match*: 664, 665, 666, 667, 668, 669. xml_parser: 646, 674, 807, 1030. *XMLASSERT*: <u>646</u>, <u>651</u>, 836. XMLAttrKind: 615, 616, 1281, 1284. XMLAttrName: 616, 712, 745, 746, 747, 760, 836, 837, 838, 839, 1196, 1197, 1199, 1201, 1202, 1206, 1215, 1216, 1222, 1223, 1224, 1225, 1226, 1227, 1228, 1229, 1230, 1231, 1232, 1239, 1242, 1243, 1244, 1245, 1246, 1247, 1248, 1250, 1251, 1253, 1258, 1259, 1267, 1271,

1275, 1280, 1281, 1288, 1290, 1291, 1294, 1297, 1298, 1299, 1300, 1301, 1303, 1305, 1306, 1307, 1308, 1310, 1311, 1313, 1314, 1316, 1319, 1322, 1323, 1324, 1325, 1326. XMLAttrObj: 649, 650.XMLAttrPtr: 649, 673, 690, 691, 1279, 1280. XMLElemKind: 613, 614, 1281, 1283. XMLElemName: 614, 745, 746, 747, 760, 836, 838, 839, 1192, 1197, 1199, 1200, 1202, 1205, 1215, 1216, 1227, 1228, 1238, 1239, 1240, 1244, 1247, 1248, 1249, 1252, 1256, 1257, 1261, 1262, 1263, 1265, 1267, 1268, 1269, 1270, 1271, 1272, 1274, 1275, 1281, 1293, 1310, 1311, 1315, 1318, 1321, 1323, 1324, 1325, 1326. XMLInStreamObj: 688, 689, 690, 691, 692, 1277. XMLInStreamPtr: 688, 807, 836, 837, 1188, 1192. XMLOutStreamObj: 693, 694, 695, 696, 697, 698, 700, 701, 702, 703, 704, 705, 706, 707, 708, 709, <u>710</u>, <u>711</u>, <u>712</u>, <u>713</u>, 760, 811, 838, 839, 1194. XMLOutStreamPtr: 693, 745. XMLParserObj: 661, 662, 663, 664, 665, 668, <u>670, 671, 672, 673, 688, 690.</u> XMLScannObj: 653, 654, 656, 657, 660, 661. XMLTokenKind: 648, 653. XMLToStr: 674, 677, 1322, 1333. xx: 216.*XXX*: 198. *X1*: 302, 303, 436, 485, 579, 583, 587, 589, 590, 591, 592, 593, 594, 595, 611, 839. X2: 302, 303, 436, 485, 579, 583, 587, 589, 590, 591, 592, 593, 594, 595, 611, 839. y: 221. *Y1*: 436, 597, 601, 609, 610, 611. Y2: 436, 597, 601, 609, 610, 611. z: 226, 624. ZeroPos: 128.zz: 216.z1: 280, 281, 282, 283, 286, 287, 288, 289, 292, 293, 294, 295, 296, 297, 304, 305. z2: 280, 281, 282, 283, 286, 287, 288, 289, 292, 293, 294, 295, 296, 297, 304, 305.

```
(Absolute value for an arbitrary-precision number 205) Used in section 198.
(Abstract base class for formulas 968)
                                           Used in section 897.
                                        Used in section 897.
(Abstract base class for terms 913)
(Abstract base class for types 957)
                                        Used in section 897.
(Abstract vocabulary object declaration 731)
                                                   Used in section 716.
\langle \text{Add } a1 \text{ and } b1 \text{ as in step 1, Eq } (211.2) \text{ 213} \rangle
                                                   Used in section 211.
(Add token to tokens buffer and iterate 784)
                                                  Used in section 782.
\langle \text{Add two negative integers, and goto } ex 240 \rangle Used in section 239.
(Adjective (abstract syntax tree) 911)
                                           Used in section 897.
(Adjective AST constructor 912) Used in section 896.
 Adjective expression (abstract syntax tree) 907
                                                        Used in section 897.
Adjective expression AST constructor 908
                                                  Used in section 896.
Aggregate term (abstract syntax tree) 936
                                                  Used in section 915.
\langle \text{Allocate a new array for } IntPairSeq, \text{ and copy the contents over } 493 \rangle
                                                                              Used in section 492.
(Allocate a new array, and copy old contents into new array 345)
                                                                        Used in section 343.
(Allocate and copy items to new sorted list 396)
                                                      Used in sections 394 and 419.
\langle Append next digit of a onto s, incrementing c 231 \rangle Used in sections 229 and 229.
 Arbitrary-precision rational arithmetic 259, 261, 263, 265, 267, 269, 271, 273, 275
(Arithmetic for arbitrary-precision integers 211, 215, 219, 221, 224, 226, 233, 235, 237, 239, 244, 249, 251, 253, 255)
    Used in section 198.
(Assert MML version is compatible with Mizar version 642)
                                                                   Used in section 639.
 Assert fCompare is present 406
                                     Used in sections 405 and 426.
\langle \text{ Basic arithmetic operations declarations 204, 225, 232, 234, 236, 238, 243, 248, 250, 252, 254} \rangle
                                                                                                Used in section 198.
(Block object implementation 854, 855, 856, 857, 858, 859, 860) Used in section 843.
 Block object interface 853 \ Used in section 848.
BlockKinds (syntax.pas) 852
                                     Used in section 848.
 Calculate (-a) + b = b - a and goto ex 241 \text{ Used in section 239.
 Calculate (-a) - (-b) and goto ex 246 Used in section 244.
 Calculate (-a) - b = -(a+b) and goto ex 245
                                                        Used in section 244.
 Calculate a + (-b) = a - b and goto ex 242 \ Used in section 239.
 Calculate difference of two positive integers 247 Used in section 244.
 Calculate quotient for nonzero divisor 297 \ Used in section 296.
 Calculate the usual multiplication of complex numbers 294
                                                                   Used in section 293.
 Carry the c_{m+1} as in step 2, Eq (211.3) 214 \( \) Used in section 211.
 Characters prohibited by MScanner 794 \ Used in section 793.
(Check all extendible items have been digested, otherwise raise error 374)
                                                                                  Used in sections 373, 375, 376, 377,
    382, and 416.
(Check every remaining open (left) parentheses has a corresponding partner 1719)
                                                                                           Used in section 1716.
 Check for 172/173 error, goto AfterBalance if erred 1599
                                                                   Used in section 1598.
 Check for 174/175 error, goto AfterBalance if erred 1600
                                                                  Used in section 1598.
 Check for errors with definition items 1433
                                                 Used in section 1429.
 Check identifier is not a number 782
                                            Used in section 773.
 Check if arbitrary-precision integers are zero 201
                                                        Used in section 198.
 Check if functor symbol is valid 720
                                          Used in section 718.
 Check if predicate symbol is valid 721 \ Used in section 718.
 Check term Bl has valid functor format, goto AfterBalance if not 1605
                                                                                  Used in section 1604.
 Check the popped item's linkages are valid 1456 \ Used in section 1429.
 Check we are redefining a mode, attribute, functor, or predicate 1889 \ Used in section 1888.
(Check Index for MCollection is within bounds, raise an error if it's out of bounds 353)
    356, and 359.
(Choice term (abstract syntax tree) 950) Used in section 915.
(Circumfix term (abstract syntax tree) 926) Used in section 915.
```

```
(Class declaration for Item object 863)
                                          Used in section 848.
(Class declarations for dicthan.pas 716)
                                             Used in section 714.
(Class for Within expression 1010) Used in section 897.
Classes for formula (abstract syntax tree) 970, 972, 974, 976, 978, 980, 982, 984, 986, 988, 990, 992, 994, 996, 998,
    1000, 1002, 1004, 1006, 1008
                                 Used in section 897.
(Classes for terms (abstract syntax tree) 915) Used in section 897.
Classes for type (abstract syntax tree) 958, 960, 962, 964, 966)
                                                                 Used in section 897.
Close parentheses for formula 1764 \ Used in section 1760.
Close the parentheses 1734
                               Used in section 1727.
Communicate with items (parser.pas) 1773) Used in section 1673.
Compare bracket symbols 823 \ Used in section 821.
Compare formats 819
                           Used in section 810.
Compare infix symbols 824
                                Used in sections 821 and 823.
Compare prefix symbols 822
                                 Used in section 821.
Compare same kinded symbols with the same number 821
                                                                 Used in section 820.
Compare symbols of the same kind 820
                                            Used in section 819.
Compare two positive integers with same number of digits 207
                                                                    Used in section 206.
(Complex-rational arbitrary-precision arithmetic 279, 281, 283, 285, 287, 289, 291, 293, 296, 299, 301, 303, 305)
    Used in section 198.
(Constants for pcmizver.pas 81, 82, 83) Used in section 80.
Constants for xml_parser.pas 647
                                        Used in section 646.
Constants for common error messages reported to console 124
                                                                     Used in section 89.
Constants of xml_dict.pas 614, 616
                                         Used in section 612.
Construct the term's syntax tree after balancing arguments among subterms 1607)
                                                                                          Used in section 1597.
Constructors for derived format classess 812, 814, 816, 818
                                                               Used in section 810.
Constructors for example statements (wsmarticle.pas) 1084
                                                                    Used in section 1081.
Consume "does not" or "do not", raise error otherwise 1749
                                                                     Used in section 1748.
Consume "per cases", raise an error if they're missing 1823)
                                                                    Used in section 1821.
Convert "-0" into zero 202
                                 Used in section 201.
Copy a and b into a1, b1 ensuring a1 is a longer string 212
                                                                   Used in section 211.
Copy a into a1 and b into b1, ensuring b1 is a shorter string 222 \quad Used in section 221.
Copy environment variable's value into lStr until we find null character 69
Copy remainder of a into s, and terminate the function 230
                                                                  Used in section 229.
Copy the variable name as a null-terminated string 68 \ Used in section 67.
Count lNbr the number of dictionary entries for an article 739
                                                                     Used in section 738.
                                                   Used in section 843.
Create a subexpression for an expression 872
(De-duplicate a string list 459)
                                  Used in section 458.
(Declare XML Attribute Object 649)
                                         Used in section 648.
Declare XML Parser object 661
                                     Used in section 648.
(Declare XML Scanner Object type 653) Used in section 648.
(Declare FileDescrCollection data type 622)
                                               Used in section 617.
Declare FileDescr data type 619
                                      Used in section 617.
\langle \text{ Declare } MBracketFormat \text{ object } 817 \rangle
                                         Used in section 809.
(Declare MFormatsList object 825)
                                       Used in section 809.
\langle \text{ Declare } MFormat \text{ object } 811 \rangle Used in section 809.
Declare MInfixFormat object 815
                                       Used in section 809.
⟨ Declare MPrefixFormat object 813⟩
                                        Used in section 809.
(Declare classes for _formats.pas 809) Used in section 807.
(Declare internal methods for StringList 453)
                                                 Used in section 452.
\langle Declare public comparison operators for arbitrary-precision numbers 302, 304\rangle Used in section 198.
(Declare public complex-valued arbitrary precision arithmetic 284, 286, 288, 290, 292, 295, 298, 300)
    section 198.
```

638 NAMES OF THE SECTIONS Mizar Parser

```
⟨ Declare Position as record 129⟩
                                                          Used in section 128.
(Delphi declaration of CtrlSignal for Windows 166)
                                                                                     Used in section 164.
⟨ Delphi implementation of InitCtrl for Windows 163⟩
                                                                                        Used in section 161.
(Determine the ID 776)
                                         Used in section 773.
                                                                          Used in section 1225.
Emit XML for Fraenkel term (WSM) 1233
Emit XML for consider contents (WSM) 1262
                                                                                 Used in section 1260.
Emit XML for reconsider contents (WSM) 1263
                                                                                    Used in section 1260.
Emit XML for aggregate term (WSM) 1229
                                                                           Used in section 1225.
Emit XML for assumptions item (WSM) 1272
                                                                               Used in section 1264.
Emit XML for attribute pattern (WSM) 1246
                                                                               Used in section 1241.
Emit XML for attributive formula (WSM) 1218
                                                                                 Used in section 1207.
(Emit XML for biconditional formula (WSM) 1212)
                                                                                     Used in section 1207.
 Emit XML for bracket functor pattern (WSM) 1244
                                                                                        Used in section 1241.
Emit XML for circumfix term (WSM) 1228
                                                                         Used in section 1225.
(Emit XML for conditional formula (WSM) 1211)
                                                                                 Used in section 1207.
(Emit XML for conjunction (WSM) 1209)
                                                                     Used in section 1207.
(Emit XML for definition-related items (WSM) 1264, 1265, 1266, 1267, 1268, 1269, 1270, 1271)
                                                                                                                                                      Used in
       section 1260.
Emit XML for disjunction (WSM) 1210
                                                                     Used in section 1207.
(Emit XML for exactly qualification term (WSM) 1236)
                                                                                            Used in section 1225.
Emit XML for existential formula (WSM) 1221
                                                                                 Used in section 1207.
(Emit XML for flexary-conjunction (WSM) 1213)
                                                                                 Used in section 1207.
Emit XML for flexary-disjunction (WSM) 1214
                                                                                 Used in section 1207.
(Emit XML for forgetful functor (WSM) 1231)
                                                                             Used in section 1225.
(Emit XML for global choice term (WSM) 1237)
                                                                               Used in section 1225.
(Emit XML for infix functor pattern (WSM) 1243) Used in section 1241.
 Emit XML for infix term (WSM) 1227 Used in section 1225.
 Emit XML for internal forgetful functor (WSM) 1232
                                                                                          Used in section 1225.
(Emit XML for mode pattern (WSM) 1245)
                                                                         Used in section 1241.
(Emit XML for multi-predicative formula (WSM) 1217)
                                                                                           Used in section 1207.
Emit XML for negated formula (WSM) 1208
                                                                             Used in section 1207.
(Emit XML for placeholder (WSM) 1226)
                                                                     Used in section 1225.
Emit XML for predicate pattern (WSM) 1242
                                                                               Used in section 1241.
Emit XML for predicative formula (WSM) 1215
                                                                                  Used in section 1207.
Emit XML for qualification term (WSM) 1235
                                                                               Used in section 1225.
Emit XML for qualifying formula (WSM) 1219
                                                                                Used in section 1207.
(Emit XML for registration-related items (WSM) 1273, 1274)
                                                                                                   Used in section 1260.
(Emit XML for right-side of predicative formula (WSM) 1216)
                                                                                                      Used in section 1207.
(Emit XML for schema (WSM) 1261)
                                                               Used in section 1260.
(Emit XML for selector term (WSM) 1230)
                                                                        Used in section 1225.
(Emit XML for simple Fraenkel term (WSM) 1234)
                                                                                    Used in section 1225.
(Emit XML for universal formula (WSM) 1220)
                                                                               Used in section 1207.
(Empty method declarations for SubexpObj 877)
                                                                                Used in section 873.
\langle Ensure b \leq s by adding another digit of a, initialize z = 228 \rangle Used in section 227.
 Ensure capacity of MCollection, raise coOverFlow error if \Delta = 0.357
\langle \text{Ensure } Count \leq ALimit \leq MaxCollectionSize 344 \rangle
                                                                                     Used in sections 343 and 419.
\langle \text{Ensure } Count < aLimit < MaxCollectionSize for sorted lists 395} \rangle
                                                                                                            Used in section 394.
\langle \text{Ensure } Count + fExtCount \leq ALimit \leq MaxCollectionSize 380} \rangle
                                                                                                           Used in section 379.
Error codes for parser 1675, 1681, 1694, 1697, 1699, 1701, 1704, 1707, 1710, 1712, 1715, 1717, 1723, 1726, 1728, 1730,
       1732, 1741, 1743, 1745, 1750, 1753, 1755, 1759, 1761, 1763, 1765, 1771, 1784, 1786, 1788, 1793, 1795, 1797, 1799, 1801, 1797, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1799, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801, 1801
       1806, 1808, 1810, 1812, 1814, 1824, 1826, 1831, 1836, 1839, 1841, 1844, 1846, 1848, 1850, 1852, 1854, 1857, 1860, 1862,
```

```
1865, 1867, 1874, 1876, 1878, 1882, 1884, 1886, 1890, 1896, 1899, 1901, 1905, 1907, 1909, 1912, 1916, 1918 Used in
    section 1674.
Exactly term (abstract syntax tree) 948
                                                Used in section 915.
Example classes (wsmarticle.pas) 1083
                                                Used in section 1080.
(ExpKinds (syntax.pas) 869)
                                    Used in section 848.
(Expression class declaration 870)
                                        Used in section 848.
(Expression constructor 871)
                                  Used in section 843.
(Extended block class declaration 1385)
                                             Used in section 1379.
(Extended block implementation 1386, 1393, 1394, 1395, 1396, 1397, 1398, 1399, 1401, 1402, 1403)
                                                                                                        Used in
    section 1380.
Extended expression class declaration 1669 Used in section 1379.
Extended expression implementation 1670, 1671
                                                       Used in section 1380.
\langle Extended item class declaration 1405\rangle Used in section 1379.
Extended item implementation 1406, 1429, 1457, 1459, 1460, 1461, 1462, 1463, 1465, 1466, 1468, 1469, 1470, 1471,
    1473, 1474, 1475, 1476, 1479, 1480, 1481, 1483, 1484, 1485, 1486, 1487, 1490, 1491, 1492, 1493, 1494, 1495, 1498, 1499,
    1501, 1502, 1503, 1504, 1505, 1506, 1507, 1509, 1510, 1512, 1513, 1514, 1515, 1517, 1519, 1521, 1522, 1523, 1524, 1525,
    1527, 1529, 1530, 1531, 1532, 1533, 1534, 1535, 1536, 1537, 1538, 1539, 1540, 1541, 1542, 1543, 1544, 1545, 1546, 1548,
    1549, 1551, 1552, 1553, 1555, 1556, 1557, 1558, 1559, 1560, 1561, 1563, 1564, 1565 Used in section 1380.
(Extended subexpression class declaration 1568) Used in section 1379.
Extended subexpression implementation 1570, 1571, 1572, 1573, 1575, 1576, 1577, 1578, 1579, 1580, 1581, 1582, 1583,
    1584,\ 1585,\ 1586,\ 1587,\ 1588,\ 1589,\ 1590,\ 1591,\ 1593,\ 1597,\ 1609,\ 1610,\ 1611,\ 1612,\ 1613,\ 1614,\ 1615,\ 1616,\ 1617,\ 1618,
    1619, 1620, 1621, 1622, 1623, 1624, 1625, 1626, 1627, 1628, 1629, 1630, 1631, 1632, 1633, 1634, 1635, 1636, 1637, 1638,
    1639, 1640, 1641, 1642, 1643, 1644, 1645, 1646, 1647, 1648, 1649, 1650, 1651, 1652, 1653, 1654, 1655, 1656, 1657, 1658,
    1659, 1660, 1661, 1662, 1663, 1664, 1665, 1666, 1667, 1668 Used in section 1380.
(Find needle in MSortedList by binary search 402) Used in section 401.
(Find needle in MSortedList by brute force 403) Used in section 401.
(Find the first significant 'sign' 795) Used in section 793.
(Flush XML output buffer, if full 699)
                                           Used in section 698.
 Forgetful functor (abstract syntax tree) 938
                                                   Used in section 915.
(Fraenkel terms (abstract syntax tree) 942, 944) Used in section 915.
(Free old arrays, if any 397) Used in sections 394 and 419.
 Free the old array, if it isn't nil 494 \ Used in section 492.
(FreePascal declaration of CtrlSignal for Windows 165)
                                                               Used in section 164.
\langle FreePascal implementation of InitCtrl for Windows 162\rangle Used in section 161.
\langle \text{GNU License 4} \rangle Used in sections 35, 80, 89, 127, 151, 153, 167, 185, 198, 307, 612, 617, 646, 674, 714, 748, 807, 841,
    879, 895, 1030, 1373, 1379, and 1672.
(Get closed subterm of Fraenkel operator or enumerated set 1702)
(Get closed subterm of bracketed expression 1700) Used in section 1695.
 Get closed subterm of choice operator 1706 \ Used in section 1695.
 Get closed subterm of identifier 1696
                                             Used in section 1695.
 Get closed subterm of structure 1698
                                             Used in section 1695.
 Get contents of XML start tag 666
                                          Used in section 665.
Get environment variable, Delphi-compatible mode 67
                                                               Used in section 66.
 Get the identifier number for current word 1381 \ Used in section 1380.
(Get the next digit for dividing arbitrary-precision integers 229) Used in section 226.
 Get token kind based off of leading character 659 \ Used in section 657.
Global variables (_formats.pas) 808)
                                            Used in section 807.
(Global variables declared in librenv.pas 618) Used in section 617.
Global variables introduced in parseraddition.pas 1382, 1390, 1392, 1407, 1411, 1414, 1420, 1423, 1427, 1436,
    1448,\ 1453,\ 1467,\ 1477,\ 1488,\ 1496,\ 1500,\ 1508,\ 1516,\ 1518,\ 1520,\ 1526,\ 1528,\ 1547,\ 1550,\ 1554,\ 1574 Used in
    section 1379.
(Global variables publicly declared in wsmarticle.pas 1187) Used in section 1030.
```

Mizar Parser

```
\langle Guess the k^{th} functor format 1604\rangle Used in section 1598.
\langle Guess MizFiles from environment variables or executable path 645 \rangle Used in section 644.
(Halt: invalid article name 76) Used in section 75.
\langle Halt: we can't open the file 71 \rangle Used in section 70.
(Handle runtime error cases for monitor.pas 156)
                                                                                    Used in section 155.
\langle Hard delete entry I 456\rangle Used in section 455.
\langle Has overflow occurred in NatFunc. Add? 545\rangle Used in section 544.
(If cannot open the FileName, report an error and halt 136)
                                                                                                   Used in section 135.
(If we matched a dictionary entry, then initialize FoundToken 780)
\langle \text{Implement } MFormatObj 838 \rangle Used in section 810.
(Implementation for I/O of XML 676, 679, 681, 685, 689, 694, 700)
                                                                                                         Used in section 674.
(Implementation for MScanner 886, 887, 888, 889, 890, 891, 892, 893)
                                                                                                            Used in section 879.
\langle \text{Implementation for } MFormatsList | 826 \rangle
                                                                    Used in section 810.
(Implementation for _formats.pas 810)
                                                                    Used in section 807.
(Implementation for dicthan.pas 718, 723, 727, 732, 734, 738, 740, 741)
                                                                                                                Used in section 714.
(Implementation for errhan.pas 131, 133, 135, 140, 142, 144, 146, 148, 150)
                                                                                                                     Used in section 127.
(Implementation for librenv.pas 620, 623, 628, 630, 632, 634, 638, 639, 643) Used in section 617.
(Implementation for mobjects.pas 308, 309) Used in section 307.
(Implementation for monitor.pas 155, 157)
(Implementation for mstate.pas 187, 189, 191, 193, 197) Used in section 185.
(Implementation for mtime.pas 168, 171, 173, 175, 177, 178, 180, 184) Used in section 167.
(Implementation for pcmizver.pas 85, 86, 87, 88) Used in section 80.
(Implementation for syntax.pas 843) Used in section 841.
(Implementation for wsmarticle.pas 1034, 1037, 1038, 1041, 1043, 1045, 1049, 1052, 1056, 1058, 1060, 1062, 1065,
       1067, 1069, 1071, 1074, 1076, 1078, 1081, 1087, 1090, 1092, 1094, 1097, 1098, 1105, 1107, 1109, 1111, 1114, 1116, 1118,
       1120, 1122, 1127, 1129, 1131, 1133, 1135, 1137, 1139, 1141, 1143, 1145, 1147, 1149, 1151, 1153, 1155, 1157, 1159, 1161,
       1163, 1165, 1168, 1170, 1172, 1174, 1176, 1178, 1180, 1182, 1184, 1185, 1186, 1188, 1193, 1195, 1196, 1197, 1198, 1199,
       1200, 1201, 1202, 1203, 1204, 1205, 1206, 1207, 1222, 1223, 1224, 1225, 1238, 1239, 1240, 1241, 1247, 1248, 1249, 1250,
       1251,\ 1252,\ 1253,\ 1254,\ 1255,\ 1256,\ 1257,\ 1258,\ 1259,\ 1260,\ 1275,\ 1278,\ 1279,\ 1280,\ 1281,\ 1282,\ 1283,\ 1284,\ 1285,\ 1286,\ 1281,\ 1282,\ 1283,\ 1284,\ 1285,\ 1286,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 1281,\ 
       1287, 1288, 1289, 1290, 1291, 1292, 1293, 1294, 1295, 1298, 1299, 1300, 1301, 1302, 1303, 1304, 1305, 1306, 1307, 1308,
       1309, 1310, 1311, 1312, 1313, 1314, 1315, 1316, 1317, 1318, 1319, 1320, 1321, 1322, 1323, 1324, 1325, 1326, 1327, 1329,
       1330, 1331, 1332, 1333, 1334, 1335, 1336, 1337, 1338, 1339, 1340, 1341, 1342, 1343, 1344, 1345, 1346, 1347, 1348, 1349,
       1350, 1351, 1352, 1353, 1354, 1355, 1356, 1357, 1358, 1359, 1360, 1361, 1362, 1363, 1364, 1365, 1366, 1367, 1368, 1369,
       1370, 1371, 1372 Used in section 1030.
(Implementation for mconsole.pas 96, 98, 100, 103, 105, 107, 109, 111, 113, 114, 116, 119, 121, 123, 126)
       sections 89 and 117.
(Implementation for scanner.pas 750, 752, 754, 755, 756, 759, 760, 762, 763, 765, 768, 769, 773, 788, 789, 791, 793, 802,
       803, 804, 805, 806 \ Used in section 748.
(Implementation of XML Parser 650, 651, 652, 654, 656, 657, 660, 662, 668, 670, 671)
(Implementation of abstract syntax 896)
                                                                     Used in section 895.
(Implementation of parser additions 1380) Used in section 1379.
(Implementation of parser.pas 1673, 1676, 1677, 1679, 1680, 1682, 1683, 1684) Used in section 1672.
(Implementing formula AST 971, 973, 975, 977, 979, 981, 983, 985, 987, 989, 991, 993, 995, 997, 999, 1001, 1003, 1005,
       1007, 1009
                           Used in section 896.
(Implementing term AST 917, 919, 921, 923, 925, 927, 929, 931, 933, 935, 937, 939, 941, 943, 945, 947, 949, 951, 953,
                Used in section 896.
(Implementing type AST 959, 961, 963, 965, 967) Used in section 896.
(Import units for mconsole.pas 90) Used in section 89.
(Incorrect term (abstract syntax tree) 954) Used in section 915.
(Index first character appearances among definitions 758) Used in section 756.
(Inference kinds (wsmarticle.pas) 1054)
                                                                      Used in section 1047.
(Infix term (abstract syntax tree) 922) Used in section 915.
```

```
(Initialize WS Mizar article 1383) Used in section 1380.
(Initialize lKind, but exit if dictionary line contains invalid symbol 719)
                                                                                     Used in section 718.
(Initialize errhan.pas state variables 137) Used in sections 135 and 140.
\langle \text{Initialize main } extBlock \text{ instance } 1388 \rangle
                                               Used in section 1386.
(Initialize "proper text" extBlock instance 1391)
                                                          Used in section 1386.
\langle Initialize default values for extBlock instance 1387\rangle Used in section 1386.
(Initialize explicit priority for functor entry in dictionary 726)
                                                                         Used in section 724.
(Initialize extended item by ItemKind 1410, 1413, 1415, 1417, 1419, 1422, 1426) Used in sections 1406 and 1424.
(Initialize identifier names from .idx file 1192) Used in section 1188.
\langle Initialize symbol priorities, determine last ll, pl values 1608\rangle
                                                                        Used in section 1607.
\langle Initialize the fields for newly allocated extItem object 1409\rangle Used in section 1406.
\langle \text{ Initialize the } ExitProc \ 195 \rangle
                                   Used in section 193.
(Initialize Bl., goto AfterBalance if term has at most one argument 1603)
(Initialize MizFiles 644) Used in section 643.
\langle \text{Initialize } To\_Right \text{ and } ArgsLength \text{ arrays } 1601, 1602 \rangle
                                                                Used in section 1598.
(Initilize explicit infinitive for a predicate entry in dictionary 725)
\langle \text{Insert } (X, 1) | 533 \rangle Used in section 532.
(Insert item to the end of the caller 393)
                                                 Used in section 392.
(Int relation implementation 498, 499, 500, 501, 502, 503, 504, 505)
                                                                          Used in section 309.
(Interface for errhan.pas 128, 130, 132, 134, 139, 141, 143, 145, 147, 149)
                                                                                Used in section 127.
(Interface for mstate.pas 186, 188, 192, 196)
                                                    Used in section 185.
(Interface for mtime.pas 172, 174, 176, 179, 183)
                                                        Used in section 167.
\langle \text{Interface for syntax.pas } 848 \rangle Used in section 841.
(Interface for abstract syntax 897) Used in section 895.
(Interface for accommodator command line options 92, 95, 99)
(Interface for other command line options 93, 104) Used in section 89.
(Interface for transfer-specific command line options 106) Used in section 89.
\langle \text{Interface for } MakeEnv \text{ command line options } 102 \rangle
                                                             Used in section 89.
(Internal forgetful functors (abstract syntax tree) 940)
                                                                 Used in section 915.
(Internal selector term (abstract syntax tree) 934) Used in section 915.
(Item object implementation 864, 865, 868)
                                                  Used in section 843.
(ItemKinds (syntax.pas) 862) Used in section 848.
(Keep parsing as long as there is an infixed operator to the right 1718) Used in section 1716.
(Load all tokens from the dictionary 757)
                                                 Used in section 756.
(Local constants for parser.pas 1674, 1678)
                                                 Used in section 1673.
(Local variables for parser additions 1389, 1408, 1412, 1418, 1421, 1424, 1425, 1428, 1432, 1438, 1440, 1442, 1449, 1451,
     1454, 1458, 1464, 1472, 1478, 1482, 1489, 1497, 1511, 1562, 1569 Used in section 1379.
\langle \text{Long division of } a \text{ by } b \text{ 227} \rangle
                                    Used in section 226.
(MScanner object class 792)
                                   Used in section 749.
(MToken object class 761)
                                 Used in section 749.
(MTokeniser class 764) Used in section 749.
(Main parse method (parser.pas) 1921) Used in section 1673.
(Mark schema proof as "skipped" 1400)
                                                Used in section 1399.
(Method declarations for Item object 867) Used in section 863.
\langle Methods implemented by subclasses of SubexpObj 876, 1567\rangle
                                                                         Used in sections 877 and 1568.
(Methods overriden by extended Item class 866, 1404)
                                                                Used in sections 867 and 1405.
(Modules used by mizenv.pas 37)
                                          Used in section 35.
\langle \text{ Multiply } i^{\text{th}} \text{ digit of } a \text{ by } y \text{ 220} \rangle Used in section 219.
(Multiply i^{\text{th}} digit of b1 to a1 and add it to r 223)
                                                             Used in section 221.
(NatSeq implementation 548)
                                    Used in section 309.
(Negated adjective AST constructor 910)
                                                 Used in section 896.
(Negated adjective expression (abstract syntax tree) 909) Used in section 897.
```

```
(Non-windows FreePascal implementation for InitCtrl 160) Used in section 159.
(Numeral term (abstract syntax tree) 920) Used in section 915.
(One-argument term (abstract syntax tree) 930) Used in section 915.
 Open mml.ini file 640 \ Used in section 639.
\langle \text{ Parse XML end tag 667} \rangle Used in section 665.
Parse XML for formula with binary connective 1296
                                                           Used in section 1295.
(Parse XML for predicate-based formula 1297) Used in section 1295.
Parse "equals" definiens 1877
                                   Used in section 1872.
\langle \text{ Parse "means" definiens 1873} \rangle Used in section 1872.
(Parse "over" and any structure arguments, if any 1861) Used in section 1858.
(Parse "proof" block 1833) Used in section 1832.
(Parse "suppose" or "case" block 1827)
                                             Used in section 1821.
(Parse "take" statement for linear reasoning 1818) Used in section 1817.
Parse "then" for linear reasoning 1820
                                            Used in section 1817.
(Parse "thus" and "hence" for linear reasoning 1819) Used in section 1817.
(Parse "equals" definition-by-cases 1879)
                                             Used in section 1877.
(Parse "means" definition-by-cases 1875)
                                            Used in section 1873.
(Parse a Fraenkel operator 1703) Used in section 1702.
(Parse a "deffunc" 1807)
                             Used in section 1805.
(Parse a "defpred" 1809) Used in section 1805.
(Parse a "reconsider" statement 1813) Used in section 1805.
(Parse a "set" constant definition 1811)
                                            Used in section 1805.
(Parse an enumerated set 1705)
                                   Used in section 1702.
(Parse ancestors of structure, if there are any 1859) Used in section 1858.
Parse arguments for attribute expression 1897 \ Used in section 1895.
Parse arguments to the right 1722
                                        Used in section 1720.
Parse bracket functor pattern 1851
                                         Used in section 1847.
(Parse collective assumption 1780)
                                      Used in section 1779.
Parse comma-separated variables for quantified variables 1740
                                                                     Used in section 1739.
(Parse conditional registration cluster 1902)
                                               Used in section 1898.
Parse contents of "suppose" block 1828
                                             Used in section 1827.
Parse definiens 1872 \rangle Used in section 1870.
Parse definitions (parser.pas) 1868, 1869, 1870, 1880, 1881, 1883, 1885, 1887, 1892, 1895, 1898, 1904, 1906, 1908,
    1910, 1911, 1913, 1914 \rightarrow Used in section 1673.
Parse equation or (possibly infixed) predicate 1756
                                                          Used in section 1752.
(Parse existential registration cluster 1903) Used in section 1898.
Parse expressions (parser.pas) 1687, 1688, 1689, 1690, 1691, 1695, 1713, 1714, 1716, 1724, 1727, 1735, 1736, 1737,
    1738, 1739, 1742, 1744, 1746, 1747, 1751, 1752, 1760, 1766, 1767, 1768, 1769, 1770, 1772 Used in section 1673.
(Parse field for the structure definition 1866) Used in section 1864.
(Parse forgetful functor or choice of structure type 1709)
                                                              Used in section 1706.
(Parse formula with "does not" or "do not" 1757) Used in section 1752.
(Parse formula with "is not" or "is not" 1758) Used in section 1752.
(Parse functor registration cluster 1900) Used in section 1898.
(Parse infix functor pattern 1849) Used in section 1847.
(Parse item for notation block 1893)
                                       Used in section 1892.
(Parse item in definition block 1888)
                                         Used in section 1887.
(Parse iterative equations 1834) Used in section 1832.
\langle \text{ Parse justification for scheme } 1920 \rangle Used in section 1915.
(Parse left arguments in a formula 1754) Used in section 1752.
(Parse library references 1794) Used in section 1792.
(Parse loci, assumptions, unexpected items in a definition block 1891) Used in section 1888.
(Parse mode as radix type 1729) Used in section 1727.
```

```
(Parse mode definition 1871) Used in section 1870.
(Parse multi-predicate with "does" or "do" in copula 1748)
                                                                 Used in section 1747.
\langle \text{ Parse next block 1923} \rangle Used in section 1922.
(Parse optional left-paren 1733) Used in section 1727.
(Parse pattern for "set" as a mode 1840)
                                             Used in section 1838.
Parse pattern for a mode symbols 1842 Used in section 1838.
(Parse patterns (parser.pas) 1835, 1837, 1838, 1843, 1845, 1847, 1853, 1856, 1858) Used in section 1673.
Parse post-qualified comma-separated list of variables 1693
                                                                  Used in section 1692.
(Parse pragmas and begins 1924)
                                     Used in section 1923.
(Parse predicate pattern 1855)
                                 Used in section 1853.
Parse private formula 1762
                                Used in section 1760.
(Parse proper text 1922)
                            Used in section 1921.
(Parse reference to scheme from MML 1798) Used in section 1796.
(Parse scheme block (parser.pas) 1915) Used in section 1673.
(Parse scheme parameters 1917) Used in section 1915.
(Parse scheme premises 1919)
                                Used in section 1915.
(Parse segment of fixed variables 1783) Used in section 1782.
Parse selector functor 1708
                                Used in section 1706.
(Parse semicolon-separated items in a notation block 1894)
                                                                Used in section 1893.
(Parse simple Fraenkel expression or "the set" 1711)
                                                         Used in section 1706.
(Parse simple justifications (parser.pas) 1791, 1796, 1800) Used in section 1673.
Parse single reference 1792 Used in section 1791.
\langle \text{ Parse singule assumption } 1781 \rangle
                                    Used in section 1779.
(Parse start of XML tag 669) Used in section 668.
Parse statement of linear reasoning 1817
                                              Used in section 1816.
(Parse statements and reasoning (parser.pas) 1802, 1803, 1804, 1805, 1815, 1816, 1821, 1830, 1832)
                                                                                                       Used in
(Parse structure as radix type 1731) Used in section 1727.
(Parse term subexpressions (parser.pas) 1720) Used in section 1736.
Parse the command-line arguments for article name and options 194
                                                                            Used in section 193.
(Parse the fields of the structure definition 1864) Used in section 1858.
(Partial Binary integer Functions 580) Used in section 309.
(Partial integer function implementation 507, 527)
                                                       Used in section 309.
(Partial integers to Pair of integers Functions 598) Used in section 309.
Pattern objects (wsmarticle.pas) 1113, 1115, 1117, 1119, 1121 \tag{Used in section 1110.}
(Placeholder term (abstract syntax tree) 918) Used in section 915.
(Pop skips remaining cases 1431) Used in section 1430.
\langle \text{Pop a "let" statement 1443} \rangle Used in section 1430.
(Pop a conclusion or regular statement 1441)
                                                 Used in section 1430.
(Pop a definition item 1444, 1445, 1446, 1447, 1450) Used in section 1430.
\langle \text{ Pop a proof step } 1434, 1435, 1437, 1439 \rangle Used in section 1430.
(Pop a registration item 1452, 1455)
                                       Used in section 1430.
Pop a token from the underlying tokens stack 790
                                                         Used in sections 789 and 789.
(Populate global variables with XML entities 1189)
                                                        Used in section 1188.
(Populate the current line 796) Used in section 795.
(Populate fList with the command-line arguments 636)
                                                             Used in sections 634 and 638.
(Populate fList with the contents of FileName and exit 635) Used in sections 634 and 638.
Predicate declarations for arbitrary-precision arithmetic 278, 280, 282
                                                                            Used in section 198.
(Prepare to read in the contents of XML file 655)
                                                      Used in section 654.
(Private functor term (abstract syntax tree) 928)
                                                      Used in section 915.
(Process "case" (local procedure) 1822)
                                            Used in section 1821.
(Process attributes (parser.pas) 1725)
                                            Used in section 1724.
```

```
(Process miscellany (parser.pas) 1774, 1775, 1776, 1777, 1778, 1779, 1782, 1785, 1787, 1789, 1790)
                                                                                                       Used in
    section 1673.
(Process post-qualified segment 1692)
                                           Used in section 1691.
(Public constants for dicthan.pas 715)
                                             Used in section 714.
(Public constants for syntax.pas 842)
                                            Used in section 848.
(Public declaration for Stream Object 680)
                                                Used in section 675.
(Public declaration for Text Stream Object 684) Used in section 675.
(Public declaration for XML Output Stream 693) Used in section 675.
(Public function declarations for dicthan.pas 717)
                                                         Used in section 714.
Public function declarations for librenv.pas 627, 629, 631, 633, 637
                                                                          Used in section 617.
(Public functions for pcmizver.pas 84) Used in section 80.
(Public interface for MScanner 880, 882, 883)
                                                  Used in section 879.
(Public interface for XML Input Stream 688) Used in section 675.
Public interface for mobjects.pas 310, 311, 312, 317, 321, 322, 349, 370, 387, 409, 423, 428, 437, 441, 449, 451, 452,
    485, 486, 497, 506, 528, 547, 552, 568, 579, 597, 611 Used in section 307.
(Public procedures for syntax.pas 849)
                                            Used in section 848.
Public procedures implementation for syntax.pas 844, 845, 846, 847
                                                                            Used in section 843.
(Public variables for syntax.pas 850) Used in section 848.
(Publicly declared constants in wsmarticle.pas 1035)
                                                             Used in section 1030.
(Publicly declared functions in wsmarticle.pas 1033)
                                                             Used in section 1030.
(Publicly declared types in wsmarticle.pas 1032, 1036, 1042, 1044, 1047, 1048, 1050, 1051, 1055, 1057, 1059, 1061,
    1064, 1066, 1068, 1070, 1073, 1075, 1077, 1080, 1086, 1089, 1091, 1093, 1095, 1096, 1104, 1106, 1108, 1110, 1126, 1128,
    1130, 1132, 1134, 1136, 1138, 1140, 1142, 1144, 1146, 1148, 1150, 1152, 1154, 1156, 1158, 1160, 1162, 1164, 1167, 1169,
    1171, 1173, 1175, 1177, 1179, 1181, 1183, 1194, 1277, 1328
                                                             Used in section 1030.
Qualified segment (abstract syntax tree) 901, 903, 905
                                                             Used in section 897.
 Qualified segment AST constructor 902, 904, 906
                                                       Used in section 896.
 Qualified term (abstract syntax tree) 946 \ Used in section 915.
 Raise error over invalid term subexpression 1721 \ Used in section 1720.
 Rational arithmetic declarations 258, 260, 262, 264, 266, 268, 270, 272, 274
                                                                              Used in section 198.
Read formats from an XML input stream 837 Used in section 810.
 Read line into vocabulary from dictionary file 737
                                                          Used in section 736.
 Rebalance the long term tree 1598 \ Used in section 1597.
(Replace end of long line with record separator 797)
                                                          Used in section 796.
(Replace every invalid character in current line with the unit character 798)
                                                                                    Used in section 796.
(Report hours and minutes 182)
                                     Used in section 180.
(Report results to command line 108, 110, 112, 115, 117, 118, 120, 122, 125)
                                                                              Used in section 89.
(Reset reserved keywords 1191)
                                    Used in section 1188.
Return false if any item of Self is in fFunc 526
                                                       Used in section 524.
 Return false if any item of fFunc is in Self 525
                                                       Used in section 524.
(Round to nearest second 181)
                                   Used in section 180.
(Scan for pragmas, and exit if we found one 800)
                                                       Used in section 796.
(Selector term (abstract syntax tree) 932) Used in section 915.
(Set current position to first line, first column 138)
                                                         Used in sections 135, 140, and 187.
(Set lResult to the index of the newly inserted string 458)
                                                                Used in section 457.
\langle Shift entries of MCollection to the left by one 354\rangle
                                                          Used in section 352.
Shift entries to the right by one 358
                                          Used in section 356.
(Simple term (abstract syntax tree) 916)
                                              Used in section 915.
(Skip all end tokens, report errors 1925)
                                              Used in section 1923.
\langle \text{Skip comments } 801 \rangle Used in section 796.
(Skip whitespace for XML parser 658)
                                          Used in section 657.
\langle Slice pragmas 774\rangle Used in section 773.
(Spell an error or EOF for the MTokeniser 767) Used in section 765.
```

```
(Spell an identifier for the MTokeniser 766)
                                                Used in section 765.
(Stacked object implementation 450)
                                         Used in section 309.
(Store XML version of vocabulary word 1190) Used in section 1189.
String collection implementation 438
                                          Used in section 309.
(String list implementation 454, 457, 460)
                                            Used in section 309.
(Subexpression constructor 874)
                                    Used in section 843.
(Subexpression destructor 875)
                                   Used in section 843.
 Subexpression object class 873
                                    Used in section 848.
(Subexpression procedures 878) Used in section 843.
(Subtract the i^{\text{th}} digit of b1 from a1 218) Used in section 215.
Swap a1 and b1 if b1 < a1 217 \ Used in section 215.
Symbol for vocabulary 722
                                Used in section 716.
(Synchronize after missing 'suppose' or 'case' token 1829)
                                                               Used in section 1827.
(Terms with arguments (abstract syntax tree) 924) Used in section 915.
 Test if one arbitrary-precision number is less than or equal to another 206, 208, 209, 210
                                                                                            Used in section 198.
 Timing utilities uses for Delphi 169
                                         Used in section 168.
 Timing utilities uses for FreePascal 170 \ Used in section 168.
 Token kinds for MScanner 884
                                    Used in section 880.
Token names for MScanner 885
                                     Used in section 882.
(Token object class 751) Used in section 749.
 Token type for MScanner 881 >
                                   Used in section 880.
 Tokens collection class 753
                                Used in section 749.
Transform XML entity into character, if encountering an XML entity at i 678
                                                                                     Used in section 677.
(Trim leading zeros from arbitrary-precision integers 200) Used in section 198.
 Trim leading zeros from numerator and denominator 203
                                                               Used in section 201.
 Trim whitespace from the right of the current line 799
                                                            Used in sections 796 and 800.
 Try to find a dictionary symbol 779 \ Used in section 773.
 Try to read the Mizar version from mml.ini 641
                                                      Used in section 639.
 Try to synchronize after failing to find initial 'case' or 'suppose' 1825 >
                                                                             Used in section 1821.
Tuples of integers 487, 488, 489, 490, 491, 492, 495, 496
(Type declarations for XML I/O 675) Used in section 674.
 Type declarations for xml_parser.pas 648 \ Used in section 646.
 Type declarations for scanner 749 \ Used in section 748.
 Types for arbitrary-precision arithmetic 256, 276
                                                      Used in section 198.
 Types of xml_dict.pas 613, 615
                                     Used in section 612.
 Units used by monitor.pas 154)
                                     Used in section 153.
 Update content of nLastWSItem based on type of item popped 1430 \rangle Used in section 1429.
 Update max arguments for structure symbol, if needed 1863
                                                                 Used in section 1858.
 Variable (abstract syntax tree) 898
                                         Used in section 897.
 Variable AST constructor 899
                                   Used in section 896.
 Variables for finishing a long term in a subexpression 1606
                                                                 Used in section 1597.
 Variables for slicing a phrase 772, 775, 778, 781, 783, 785
                                                            Used in section 773.
(Vocabulary object declaration 733) Used in section 716.
Weakly strict Item class 1040 Used in section 1036.
Whoops! ID turns out to be invalid, insert an error token, then continue 777
                                                                                   Used in section 776.
 Whoops! Identifier turned out to be a number! 786
                                                        Used in section 782.
Whoops! We found an unknown token, insert a 203 error token 787
\langle Windows implementation for CtrlSignal\ 164 \rangle Used in section 161.
\langle Windows implementation for InitCtrl 161\rangle Used in section 159.
Within expression AST implementation 1011, 1012, 1013, 1014, 1015, 1016, 1017, 1018, 1019, 1020, 1021, 1022, 1023,
    1024, 1025, 1026, 1027, 1028, 1029
                                     Used in section 896.
Write symbols to vocabulary XML file 747 Used in section 745.
```

```
Write vocabulary counts to XML file 746
                                                            Used in section 745.
(Zero and units for arbitrary-precision 257, 277)
                                                                  Used in section 198.
\langle format.pas 807 \rangle
\langle base/librenv.pas 617 \rangle
\langle base/xml_parser.pas 646 \rangle
 xml_dict.pas 612 >
 "Borrow 1" procedure for \_Sub \ 216
                                                     Used in section 215.
 "It" term (abstract syntax tree) 952
                                                      Used in section 915.
 abstract_syntax.pas 895
\langle dicthan.pas 714 \rangle
\langle \text{errhan.pas } 127 \rangle
\langle \text{ implementation of mizenv. pas } 38, 40, 42, 44, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 70, 72, 73, 74, 75, 77, 78 \rangle
                                                                                                                                      Used in
     section 35.
\langle \text{ info.pas } 151 \rangle
(interface for mizenv.pas 36, 39, 41, 43, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65)
                                                                                                Used in section 35.
\langle \text{ mconsole.pas } 89 \rangle
\langle \text{ mizenv.pas } 35 \rangle
\langle \text{ mobjects.pas } 307 \rangle
\langle \text{ monitor.pas } 153 \rangle
\langle \text{ mstate.pas } 185 \rangle
\langle \text{ mtime.pas } 167 \rangle
\langle \text{ numbers.pas } 198 \rangle
(parser.pas 1672)
(parseraddition.pas 1379)
\langle \text{ pcmizver.pas } 80 \rangle
\langle \text{ pragmas.pas } 1373 \rangle
\langle \text{ scanner.pas } 748, 879 \rangle
(syntax.pas 841)
(wsmarticle.pas 1030)
\langle xml_{inout.pas} 674 \rangle
 DisplayLine global constant 91 >
                                                Used in section 108.
\langle IntSequence implementation 553 \rangle
                                                 Used in section 309.
\langle IntSet \text{ Implementation } 569 \rangle
                                          Used in section 309.
\langle MCollection \text{ implementation } 350, 351, 352, 355, 356, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369 \rangle
     section 309.
(MExtList implementation 371, 372, 373, 375, 376, 377, 378, 379, 381, 382, 383, 384, 385, 386)
                                                                                                                     Used in section 309.
\langle MIntCollection implementation 442 \rangle
                                                     Used in section 309.
\langle MList \text{ implementation } 324 \rangle
                                          Used in section 309.
 MObject implementation 313 >
                                             Used in section 309.
\langle MSortedCollection implementation 429 \rangle
                                                         Used in section 309.
(MSortedExtList implementation 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422)
                                                                                                                        Used in section 309.
(MSortedList implementation 388, 389, 390, 391, 392, 394, 398, 400, 401, 404, 405, 407, 408)
                                                                                                                  Used in section 309.
 MSortedStrList implementation 424
                                                      Used in section 309.
\langle MStrObj \text{ implementation 318} \rangle Used in section 309.
\langle NatFunc \text{ implementation 529, 534, 546} \rangle Used in section 309.
```

Mizar Parser

		Section	Page
0	Introduction	1	1
	1. Mizar's workflow	16	4
	1. Map of Mizar	19	5
	2. Log of todos, bugs, improvements		7
	3. Review of Pascal	33	11
1	Mizar environment	35	12
2	PC Mizar Version	80	21
3	Mizar Console	89	24
	1. Parsing command-line arguments	92	25
	2. Reporting results to the console	108	29
4	Error handling	127	33
5	Info file handling	151	37
6	Monitor	153	39
7	Time utilities	167	43
8	Mizar internal state		47
9	Arbitrary precision arithmetic	198	49
	1. Arbitrary-precision integers		50
	1. Arithmetic operations	211	53
	2. Arbitrary-precision rational arithmetic		63
	3. Rational complex numbers	276	66
	1. Arithmetic operations	284	67
	4. Comparison functions	302	70
10	Mizar Objects and Data Structures	306	71
	1. Base object	312	74
	2. Mizar String Object		76
	3. Mizar List	319	77
	4. Mizar Collection Class	349	85
	5. Simple Stacked (Extendible) Lists		90
	6. Sorted lists	387	95
	7. Sorted extendible lists	409	104
	8. Sorted list of strings	423	109
	9. Sorted collections	428	110
	10. String collection	437	113
	11. Int collections	441	114
	12. Stacked list of objects	449	116
	13. String list	451	117
	14. Tuples of integers	485	125
	15. Relations of Integers as finite sets	497	128
	16. Finite sets of integers		131
	17. Function of natural numbers		136
	18. Sequences of natural numbers		143
	19. Integer sequences		144
	20. Integer sets	568	148
	21. Partial Binary integer Functions		151
	22. Partial integers to Pair of integers Functions		156
11	YMI Dictionary	619	160

Mizar Parser	TABLE OF CONTENTS	1
Wilzer T droct	TIBLE OF CONTENTS	_

12	Environment library	617 162
	1. File descriptors	619 163
	2. Collection of file descriptions	622 164
	3. Check compatibility of Mizar with MML	639 168
13	XML Parser	646 170
	1. XML Scanner Object	653 173
	2. XML Parser	661 176
14	I/O with XML	674 181
15	Vocabulary file dictionaries	714 190
16	Scanner	748 199
	1. Collections of tokens	753 201
	2. Mizar token objects	
	3. Tokeniser	
	4. Scanner Object	
17	Format	
	1. List of formats	
18	Syntax	
	1. Block Object	
	2. Item objects	
	3. Expressions	
	4. Subexpressions	
19	MScanner	
20	Abstract Syntax	
	1. Terms (abstract syntax tree)	
	2. Types (abstract syntax tree)	
	3. Formulas (abstract syntax tree)	
	4. Within expressions (deferred)	
21	Weakly strict Mizar article	
	1. Weakly strict text proper	
	2. Schemes	
	3. Private definitions	
	4. Changing types	
	5. Proof steps	
	6. Structures	
	7. Patterns	
	8. Definitions	
	9. Registrations	
	10. Helper functions	
	11. Writing WSM XML files	
	1. Emitting XML for types	
	2. Emitting XML for formulas	
	3. Emitting XML for Terms	
	4. Emitting XML for text items	
	12. Reading WSM files (deferred)	
	1. Parsing types	
	2. Parsing formulas	
	3. Parsing terms	-
	4. Parsing text items	
	13. Prettyprinting WSM files (deferred)	
22	Detour: Pragmas	
23	Detour: Parser additions	
20	1. Extended block class	
	1. Liaudided Dioca Ciabs	. 1004 406

	2	TABLE OF CONTENTS	Mizar I	Parser
	2.	Extended item class	1404	458
		1. Constructor	1406	459
		2. Popping	1429	463
		3. Registrations and notations	1457	469
		4. Processing definitions	1466	471
		5. Processing remaining statements	1515	481
	3.	Extended subexpression class	1566	494
		1. Parsing Types	1587	499
		2. Parsing operator precedence	1592	501
		3. Processing subexpressions		508
		4. Parsing formulas		514
	4.	Extended expression class	1669	523
24	Par	rser		524
	1.	Expressions	1686	528
		1. Terms	1687	528
		2. Types and Attributes	1724	537
		3. Formulas	1739	540
	2.	Communication with items	1773	548
	3.	Miscellaneous	1774	549
	4.	Simple justifications	1791	553
	5.	Statements and Reasonings	1802	556
	6.	Patterns	1835	564
	7.	Definitions	1868	570
	8.	Scheme blocks	1915	582
	9.	Main parse procedure	1921	584
	Inde	ex		586