Chapter 0

Introduction

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

We will begin with the `esmprocessor.pas` file, and all the dependencies needed to compile it. For clarity (or at least ease of reference) each "section" appearing in the table of contents corresponds to a different file.

This discusses Mizar's source code as of Git commit `9e814a9568cfb44253d677e5209c360390fe6437` (dated 2023 October 11).

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

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

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

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

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

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

4. The Mizar program is released under the GNU license. So let us place this license in one place early on.

⟨ GNU License 4 ⟩ ≡

{ This file is part of the Mizar system. Copyright (c) Association of Mizar Users. License terms: GNU General Public License Version 3 or any later version. }

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

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

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

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

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

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

format *unit* ≡ *program*
format *interface* ≡ *const*
format *implementation* ≡ *const*
format *uses* ≡ *const*

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

format *object* ≡ *record*
format *constructor* ≡ *function*
format *destructor* ≡ *function*

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

format *shr* ≡ *div*

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

define *othercases* ≡ *others*: { default for cases not listed explicitly }
define *endcases* ≡ **end** { follows the default case in an extended **case** statement }
format *othercases* ≡ *else*
format *endcases* ≡ *end*

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

define *mdebug* ≡ `@{@&$`*IFDEF MDEBUG*`@}`
define *end_mdebug* ≡ `@{@&$`*ENDIF*`@}`
format *mdebug* ≡ *begin*
format *end_mdebug* ≡ *end*

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

define *if_def*(#) ≡ `@{@&$`*IFDEF*`#@}`
define *if_not_def*(#) ≡ `@{@&$`*IFNDEF*`#@}`
define *else_if_def*(#) ≡ `@{@&$`*ELSEIF DEFINED*(#)`@}`
define *else_def* ≡ `@{@&$`*ELSE*`@}`
define *endif* ≡ `@{@&$`*ENDIF*`@}`
define *end_if* ≡ **endif**
format *if_def* ≡ *if*
format *if_not_def* ≡ *if*
format *else_if_def* ≡ *else*
format *else_def* ≡ *else*
format *endif* ≡ *end*
format *end_if* ≡ *end*

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

define *disable_io_checking* ≡ `@{@&$`$I-$`@}`
define *enable_io_checking* ≡ `@{@&$`$I+$`@}`
define *without_io_checking*(#) ≡ *disable_io_checking*; #; *enable_io_checking*

12. References. I have inline citations to the literature, but there's some references worth explicitly drawing the reader's attention to (which may or may not make it to an inline citation):

(1) Andrzej Trybulec, "Some Features of the Mizar Language", ESPRIT Workshop, Torino, 1993. Eprint: `mizar.uwb.edu.pl/project/trybulec93.pdf` — §4 discusses grammatical aspects of Mizar
(2) Freek Wiedijk, "Mizar's Soft Type System". In K. Schneider and J. Brandt, eds., *Theorem Proving in Higher Order Logics. TPHOLs 2007*, Springer, `doi:10.1007/978-3-540-74591-4_28` (Eprint pdf).
(3) Adam Grabowski, Artur Korniłowicz, and Adam Naumowicz's "Mizar in a Nutshell" (`doi:10.6092/issn.1972-5787/1980`)
(4) Artur Korniłowicz's "Registrations vs Redefinitions in Mizar" (in A. Kohlhase, P. Libbrecht, BR. Miller, A. Naumowicz, W. Neuper, P. Quaresma, F. Wm. Tompa, M. Suda (eds) *Joint Proc. FM4M, MathUI, and ThEdu*, 2016, pp.17–20, `ceur-ws.org/Vol-1785/F5.pdf`)
(5) Artur Korniłowicz's "On rewriting rules in Mizar" (*J. Autom. Reason.* **50** no.2 (2013) 203–210, `doi:10.1007/s10817-012-9261-6`)
(6) Mario Carneiro, "Reimplementing Mizar in Rust". Eprint `arXiv:2304.08391`, see especially the first two sections for an overview of Mizar's workflow. (The code is available at `github.com/digama0/mizar-rs`.)

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.

Section 0.1. MIZAR'S WORKFLOW

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

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

(1) Parser (transforms input into an abstract syntax tree, writes it to an XML file);

(2) MSM (transforms the abstract syntax tree into an explicitly typed intermediate representation) — `base/first_identification.pas`, the *MSMAnalyzer* procedure; this will require transcribing `kernel/limits.pas` (which is mostly just a bunch of constant parameters);

(3) Analyzer (performs type checking, tracks the goals, and other miscellaneous jobs) — the *Analyze* procedure in `kernel/analyzer.pas`; this requires transcribing kernel code (`lexicon.pas`, `inout.pas`, `iocorrel.pas`, `correl.pas`, `generato.pas`, `builtin.pas`, `justhan.pas`, `enums.pas`, `formats.pas`, `identify.pas`) and base code (`xmldict.pas`), approximately 19590 lines (16764 lines of code, the rest is whitespace and comments)

(4) Checker (performs the proof checking for validity) — the *InferenceChecker* procedure in `kernel/checker.pas`. This requires transcribing kernel files (`checker.pas prechecker.pas equalizer.pas unifier.pas justhan.pas`), approximately 8191 lines of code.

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

14. Accommodator. This will produce, among other outputs, the "`.dct`" file (and its XML counterpart, the "`.dcx`" file). The "`.dct`" file contains all the identifiers imported from other articles and reserved keywords for Mizar. The Tokeniser needs it to properly tokenise an article.

15. Parsing phase. We can look at `kernel/verfinit.pas` to find the parsing phase of the Mizar program is handled by the following lines of code:

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

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

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

Section 0.2. LOG OF TODOS, BUGS, IMPROVEMENTS

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

17. Possible improvements.

(1) In quicksort, picking the pivot is done by $P \leftarrow (\mathit{Low} + \mathit{High})/2$, but it should be done by $P \leftarrow \mathit{Low} + ((\mathit{High} - \mathit{Low})/2)$ 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 $P = (3 * \mathit{Low} + \mathit{High})/4$, $\mathit{P2} \leftarrow (\mathit{Low} + \mathit{High})/2$, $\mathit{P3} \leftarrow (\mathit{Low} + 3 * \mathit{High})/4$. This will improve the performance of quicksort.

(4) In §152, *GCD* could be optimized to avoid calculating $\mathit{Mul}(i, i)$ in every loop iteration.

(5) In §326, *MStringList.ObjectOf* has duplicate code.

(6) It seems that parsing Mizar text, emitting XML, and parsing XML seem to contain a lot of code which could be autogenerated from a grammar (a hypothetical "`.ebnf`" file). This would avoid duplicate work.

18. Possible bugs. I have been working through the source code with the mindset of, "How can I possibly break this?" This has led me to identify a number of situations where things can "go badly". But they are not all bugs (some are impossible to occur).

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

19. To do list. There are some things I should revisit, revise, and edit — specifically about this running commentary (*not* the Mizar source code).

(1) [Missing transcription] I skipped over transcribing the *ItemName* and other constants from `wsmarticle.pas`, which I should probably include.
(2) [Revise] The XML schema should use the `doc/mizar/xml/Mizar.rnc` schema snippets.
(3) [Revise] Make an introduction to dynamic arrays as a data structure, just to standardize the terminology used. (Make sure I stick to the standardized terminology!) Including pictures may help...
(4) [Revise] Review quicksort. I should prove that it works, too. (Has this been done in Mizar? `exchsort` seems to be the closest match.)
(5) [Improve] Give a "big picture" summary of the architecture. For example, the most interesting routine in parsing Mizar, well, it's all handled in *MTokeniser.SliceIt* (§612).
(6) [Linting] Standardize the names of basic data types. PASCAL accepts *integer* as synonymous with *Integer*, but they give different index entries.
(7) [Cosmetics] Check the typography is correct for the code
(8) [Cosmetics] Create more `WEB` macros for conditional compilation
(9) [Cosmetics] Would it help to include more UML class diagrams?
(10) [Improve] It may be useful to use UML State diagrams to explain the parser — or it may be a huge distraction?

20. Formatting types. This is still a finicky aspect of `WEB`. Strings are a type in Free PASCAL, like *Boolean*.

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

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

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

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

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

File 1

Mizar environment

23. We will need some basic library of functions. For example, trimming whitespace from a String.

⟨ mizenv.pas 23 ⟩ ≡
  ⟨ GNU License 4 ⟩
unit *mizenv*;
  interface
    ⟨ interface for mizenv.pas 24 ⟩
  implementation
    ⟨ Modules used by `mizenv.pas` 25 ⟩
    ⟨ implementation of mizenv.pas 26 ⟩
  end .

24. The interface contains all the variables for the unit, and the public facing functions and procedures.

⟨ interface for mizenv.pas 24 ⟩ ≡
var *MizFileName*, *EnvFileName*, *ArticleName*, *ArticleID*, *ArticleExt*: *String*;
procedure *SetStringLength*(**var** *aString* : *String*; *aLength* : *integer*);

See also sections 27, 30, and 34.

This code is used in section 23.

25. The implementation begins with various "uses".

⟨ Modules used by `mizenv.pas` 25 ⟩ ≡
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 23.

26. 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 26 ⟩ ≡
procedure *SetStringLength*(**var** *aString* : *String*; *aLength* : *integer*);
  var I, L: *integer*;
  begin $L \leftarrow length(aString)$;
  if $aLength \le L$ **then** $Delete(aString, aLength + 1, L - aLength)$
  else for $I \leftarrow 1$ **to** $aLength - L$ **do** $aString \leftarrow aString +$ ´␣´;
  end;

See also sections 28, 31, 35, and 46.

This code is used in section 23.

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

⟨ interface for mizenv.pas 24 ⟩ +≡
function *TrimStringLeft*(*aString* : *String*): *String*;
function *TrimStringRight*(*aString* : *String*): *String*;
function *TrimString*(**const** *aString*: *String*): *String*;

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

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

⟨ implementation of mizenv.pas 26 ⟩ +≡
function *TrimStringLeft*(*aString* : *String*): *String*;
  begin while $(length(aString) > 0) \land (aString[1] = \text{'␣'})$ **do** *Delete*(*aString*, 1, 1);
  TrimStringLeft ← *aString*;
  end;

function *TrimStringRight*(*aString* : *String*): *String*;
  begin while $(length(aString) > 0) \land (aString[length(aString)] = \text{'␣'})$ **do**
    Delete(*aString*, *length*(*aString*), 1);
  TrimStringRight ← *aString*;
  end;

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

function *TrimString*(**const** *aString*: *String*): *String*;
  begin *TrimString* ← *TrimStringRight*(*TrimStringLeft*(*aString*));
  end;

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

⟨ interface for mizenv.pas 24 ⟩ +≡
function *UpperCase*(**const** *aStr*: *String*): *String*;
function *MizLoCase*(*aChar* : *char*): *char*;
function *LowerCase*(**const** *aStr*: *String*): *String*;
function *IntToStr*(*aInt* : *integer*): *String*;

31. Now, uppercase letters.

⟨ implementation of mizenv.pas 26 ⟩ +≡
function *UpperCase*(**const** *aStr*: *String*): *String*;
  var *k*: *integer*; { index ranging over *aStr* }
    lStr: *String*; { the uppercased String being built and returned }
  begin *lStr* ← *aStr*;
  for $k \leftarrow 1$ **to** *length*(*aStr*) **do** *lStr*[*k*] ← *UpCase*(*aStr*[*k*]);
  UpperCase ← *lStr*;
  end;

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

function *MizLoCase*(*aChar* : *char*): *char*;
  begin if *aChar* ∈ [´A´ .. ´Z´] **then** *MizLoCase* ← *Chr*(*Ord*(´a´) + *Ord*(*aChar*) − *Ord*(´A´))
  else *MizLoCase* ← *aChar*;
  end;

function *LowerCase*(**const** *aStr*: *String*): *String*;
  var *i*: *integer*; { index ranging over *aStr*'s length }
    lStr: *String*; { result being built up }
  begin *lStr* ← *aStr*;
  for *i* ← 1 **to** *length*(*aStr*) **do** *lStr*[*i*] ← *MizLoCase*(*aStr*[*i*]);
  LowerCase ← *lStr*;
  end;

33. We also want to convert an integer to a String.

function *IntToStr*(*aInt* : *integer*): *String*;
  var *lStr*: *String*;
  begin *Str*(*aInt*, *lStr*); *IntToStr* ← *lStr*;
  end;

34. File name manipulation. We will want to test if a file exists, or split a path (represented as a String) into a directory and a filename.

⟨ interface for mizenv.pas 24 ⟩ +≡
function *MFileExists*(**const** *aFileName*: *String*): *boolean*;
procedure *EraseFile*(**const** *aFileName*: *String*);
procedure *RenameFile*(**const** *aName1*, *aName2*: *String*);
function *GetFileTime*(*aFileName* : *String*): *Longint*;
procedure *SplitFileName* (**const** *aFileName*: *String*;
    var *aDir*, *aName*, *aExt*: *String*) ;
function *TruncDir*(**const** *aFileName*: *String*): *String*;
function *TruncExt*(**const** *aFileName*: *String*): *String*;
function *ExtractFileDir*(**const** *aFileName*: *String*): *String*;
function *ExtractFileName*(**const** *aFileName*: *String*): *String*;
function *ExtractFileExt*(**const** *aFileName*: *String*): *String*;
function *ChangeFileExt*(**const** *aFileName*, *aFileExt*: *String*): *String*;

35. 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.

⟨ implementation of mizenv.pas 26 ⟩ +≡
function *MFileExists*(**const** *aFileName*: *String*): *boolean*;
  begin *MFileExists* ← *FileExists*(*aFileName*); **end**;

procedure *EraseFile*(**const** *aFileName*: *String*);
  begin *SysUtils.DeleteFile*(*aFileName*); **end**;

36. We will destructively rename a file. If a file with the name already exists, we delete it.

procedure *RenameFile*(**const** *aName1*, *aName2*: *String*);
  begin if *MFileExists*(*aName1*) **then** *EraseFile*(*aName2*);
  SysUtils.RenameFile(*aName1*, *aName2*);
  end;

37. Again, relying on Free Pascal's *FileAge* function.

function *GetFileTime*(*aFileName* : *String*): *Longint*;
  begin *GetFileTime* ← *FileAge*(*aFileName*); **end**;

38. Split a file name into components, namely (1) the directory, (2) the file name, (3) its extension. For example, `/path/to/my/file.exe` will be split into `/path/to/my/`, `file`, and `exe`.

This implementation depends on the compiler used (Delphi or Free Pascal).

procedure *SplitFileName* (**const** *aFileName*: *String*; { input }
  var *aDir*, *aName*, *aExt*: *String*) ; { output }
  begin
  if_def (*FPC*)
  aDir ← *SysUtils*.*ExtractFilePath*(*aFileName*);
  aName ← *SysUtils*.*ExtractFileName*(*aFileName*);
  aExt ← *SysUtils*.*ExtractFileExt*(*aFileName*);
  endif
  if_def (*DELPHI*)
  aDir ← *TPath*.*GetDirectoryName*(*aFileName*);
  aName ← *TPath*.*GetFileName*(*aFileName*);
  aExt ← *TPath*.*GetExtension*(*aFileName*);
  endif
  end ;

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

function *TruncDir*(**const** *aFileName*: *String*): *String*;
  var *Dir*, *lName*, *Ext*: *String*;
  begin *SplitFileName*(*aFileName*, *Dir*, *lName*, *Ext*); *TruncDir* ← *lName* + *Ext*;
  end;

40. "Truncating the extension" means throwing away the extension part of a path.

function *TruncExt*(**const** *aFileName*: *String*): *String*;
  var *Dir*, *lName*, *Ext*: *String*;
  begin *SplitFileName*(*aFileName*, *Dir*, *lName*, *Ext*); *TruncExt* ← *Dir* + *lName*;
  end;

41. Extracting the file directory will return *just* the directory part of a path.

function *ExtractFileDir*(**const** *aFileName*: *String*): *String*;
  var *Dir*, *lName*, *Ext*: *String*;
  begin *SplitFileName*(*aFileName*, *Dir*, *lName*, *Ext*); *ExtractFileDir* ← *Dir*;
  end;

42. Extracting the file name will throw away both the directory and extension.

function *ExtractFileName*(**const** *aFileName*: *String*): *String*;
  var *Dir*, *lName*, *Ext*: *String*;
  begin *SplitFileName*(*aFileName*, *Dir*, *lName*, *Ext*); *ExtractFileName* ← *lName*;
  end;

function *ExtractFileExt*(**const** *aFileName*: *String*): *String*;
  var *Dir*, *lName*, *Ext*: *String*;
  begin *SplitFileName*(*aFileName*, *Dir*, *lName*, *Ext*); *ExtractFileExt* ← *Ext*;
  end;

43. Changing a file name's extension. See:
`https://www.freepascal.org/docs-html/rtl/sysutils/changefileext.html`

function *ChangeFileExt*(**const** *aFileName*, *aFileExt*: *String*): *String*;
  begin *ChangeFileExt* ← *SysUtils.ChangeFileExt*(*aFileName*, *aFileExt*); **end**;

44. Getting an environmental variable.

function *GetEnvStr*(*aEnvName* : *String*): *String*;
  if_def (*FPC*)
    begin *GetEnvStr* ← *GetEnv*(*aEnvname*); **end**;
  endif
  if_def (*DELPHI*) ⟨ Get environment variable, Delphi-compatible mode 45 ⟩
  endif

45. The Delphi-compatible version of obtaining an environment variable is rather involved, so let us study it in silent meditation.

⟨ Get environment variable, Delphi-compatible mode 45 ⟩ ≡
const *cchBuffer* = 255;
var *lName*, *lpszTempPath*: **array** [0 . . *cchBuffer*] **of** *char*;
  i: *integer*; *lStr*: *String*;
  begin *lStr* ← ´´;
  for *i* ← 1 **to** *length*(*aEnvname*) **do** *lName*[*i* − 1] ← *aEnvname*[*i*];
  lName[*length*(*aEnvname*)] ← #0;
  if *GetEnvironmentVariable*(*lName*, *lpszTempPath*, *cchBuffer*) > 0 **then**
    begin for *i* ← 0 **to** *cchBuffer* **do**
      begin if *lpszTempPath*[*i*] = #0 **then** *break*;
      lStr ← *lStr* + *lpszTempPath*[*i*];
      end;
    end;
  GetEnvStr ← *lStr*;
  end;
    { restored for DELPHI4 compatibility ;-(begin GetEnvStr:=GetEnvironmentVariable(aEnvname); end; }

This code is used in section 44.

46. The *DrawMessage* and *DrawIOResult* are procedures in `mconsole.pas` (see §74, and §75).

⟨ implementation of mizenv.pas 26 ⟩ +≡
procedure *FileExam*(**const** *aFileName*: *String*);
  var
  Source: **file** ;
  I: *byte*;
    begin if *aFileName* = ´´ **then**
      begin *DrawMessage*(`´Can´´t␣open␣´´␣.miz␣´´´, ´´`); *halt*(1);
      end;
    FileMode ← 0; *Assign*(*Source*, *aFileName*); *without_io_checking*(*Reset*(*Source*)); *I* ← *ioresult*;
    if *I* ≠ 0 **then** *DrawIOResult*(*aFileName*, *I*);
    close(*Source*); *FileMode* ← 2;
    end;

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

Again, *DrawMessage* comes from `mconsole.pas` (§74).

```
procedure EnvFileExam(const aFileExt: String);
  begin if ¬MFileExists(EnvFileName + aFileExt) then
    begin DrawMessage('Can''t␣open␣''␣' + EnvFileName + aFileExt + '␣''', ''); Halt(1);
    end;
  end;
```

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

```
procedure GetFileName(ParamNr : byte; DefaultExt : String; var aFileName : String);
  var lFileExt: String;
  begin if ParamNr ≤ ParamCount then
    begin aFileName ← ParamStr(ParamNr); lFileExt ← ExtractFileExt(aFileName);
    if lFileExt = '' then aFileName ← ChangeFileExt(aFileName, DefaultExt);
    exit
    end;
  aFileName ← '';
  end;
```

49. This procedure will take the *Nr* command line argument. If it lacks a file extension, then it will append the *DefaultExt* to it. At the end, this will populate *aFileName* and *aFileExt* based on the command line.

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

```
procedure GetFileExtName(Nr : byte; DefaultExt : String; var aFileName : String; var aFileExt : String);
  begin if Nr ≤ ParamCount then
    begin aFileName ← ParamStr(Nr); aFileExt ← ExtractFileExt(aFileName);
    if aFileExt = '' then aFileExt ← DefaultExt
    else aFileName ← ChangeFileExt(aFileName, '');
    exit
    end;
  aFileName ← ''; aFileExt ← '';
  end;
```

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

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

```
procedure GetMizFileName(aFileExt : String);
  var i: integer;
  begin MizFileName ← ''; ArticleName ← ''; ArticleExt ← ''; EnvFileName ← '';
  for i ← 1 to ParamCount do
    if ParamStr(i)[1] ≠ '-' then
      begin MizFileName ← ParamStr(i); GetFileExtName(i, aFileExt, MizFileName, ArticleExt);
      ArticleName ← ExtractFileName(MizFileName); ArticleID ← UpperCase(ArticleName);
      if ¬IsMMLIdentifier(ArticleName) then
        begin DrawMessage('Only␣letters,␣numbers␣and␣_␣allowed␣in␣Mizar␣file␣names', '');
        halt(1);
        end;
      EnvFileName ← MizFileName; exit;
      end;
  end;
```

51. We will provide a standard way to refer to the article.

procedure *GetArticleName*;
  begin *GetMizFileName*(´.miz´);
  end;

52. The *EnvFileName* is populated here provided *MizFileName* is blank.

procedure *GetEnvironName*;
  var i, c: *integer*;
  begin if $MizFileName =$ ´´ **then** *GetArticleName*;
  $EnvFileName \leftarrow MizFileName$; $c \leftarrow 0$;
  for $i \leftarrow 1$ **to** *ParamCount* **do**
    if $(ParamStr(i)[1] \neq$ ´-´$)$ **then**
      begin *inc*(c);
      if $c = 2$ **then** $EnvFileName \leftarrow ParamStr(i)$;
      end;
  end;

53. The valid characters which can appear in a Mizar article name (an "MML Identifier") are uppercase Latin letters (A-Z), lowercase Latin letters (a-z), decimal digits (0-9), and underscores (_).

function *IsMMLIdentifier*(**const** *aID*: *String*): *boolean*;
  const *Allowed*: **array** $[chr(0)\,..\,chr(255)]$ **of**
      byte = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
  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;

File 2

Mizar Console

54. The Mizar Console unit is used for interacting with the command line.

⟨ mconsole.pas 54 ⟩ ≡
  ⟨ GNU License 4 ⟩
unit *mconsole*;
  interface ⟨ Interface for mconsole.pas 56 ⟩
  implementation
    ⟨ Import units for `mconsole.pas` 55 ⟩
    ⟨ Implementation for mconsole.pas 65 ⟩
  end

55. We import two modules, *pcmizver* (which we have yet to see), and *mizenv* (which we have already introduced).

⟨ Import units for `mconsole.pas` 55 ⟩ ≡
uses *pcmizver*, *mizenv*;

This code is used in section 54.

56. The interface contains the publicly available functions, as well as some specific variables for the state of the analyzer, etc.

⟨ Interface for mconsole.pas 56 ⟩ ≡
  ⟨ Report results to command line 57 ⟩
  ⟨ Constants for common error messages reported to console 60 ⟩
  ⟨ Interface for accommodator command line options 61 ⟩
  ⟨ Interface for *MakeEnv* command line options 62 ⟩
  ⟨ Interface for transfer-specific command line options 63 ⟩
  ⟨ Interface for other command line options 64 ⟩

This code is used in section 54.

57. ⟨ Report results to command line 57 ⟩ ≡
procedure *InitDisplayLine*(**const** *aComment*: *String*);
procedure *NoDisplayLine*(*fLine*, *fErrNbr* : *integer*);
  ⟨ *DisplayLine* global constant 58 ⟩
procedure *DrawMizarScreen*(**const** *aApplicationName*: *String*);
procedure *DrawArticleName*(**const** *fName*: *String*);
procedure *DrawStr*(**const** *aStr*: *String*);
procedure *FinishDrawing*;

See also section 59.

This code is used in section 56.

58. ⟨ *DisplayLine* global constant 58 ⟩ ≡
const *DisplayLine*: **procedure** (*fLine*, *fErrNbr* : *integer*) = *NoDisplayLine*;

This code is used in section 57.

59. Common routines for "drawing" output to the console.

⟨ Report results to command line 57 ⟩ +≡
procedure *EmptyParameterList*;
procedure *Noise*;
procedure *DrawPass*(**const** *aName*: *String*);
procedure *DrawTime*(**const** *aTime*: *String*);
procedure *DrawVerifierExit*(**const** *aTime*: *String*);
procedure *DrawMessage*(**const** *Msg1*, *Msg2*: *String*);
procedure *BugInProcessor*;
procedure *DrawIOResult*(**const** *FileName*: *String*; *I*: *byte*);
procedure *DrawErrorsMsg*(*aErrorNbr* : *integer*);
procedure *DisplayLineInCurPos*(*fLine*, *fErrNbr* : *integer*);

60. We also have a constant for error messages commonly encountered.

⟨ Constants for common error messages reported to console 60 ⟩ ≡
const *ErrMsg*: **array** [1 . . 6] **of** *String*[20] =
  (´´,
  ´File␣not␣found´,
  ´Path␣not␣found´,
  ´Too␣many␣open␣files´,
  ´Disk␣read␣error´,
  ´Disk␣write␣error´);

This code is used in section 56.

61. Now, we have accommodator specific options.

⟨ Interface for accommodator command line options 61 ⟩ ≡
  { Accommodator specific options: }
var *InsertHiddenFiles*, *VocabulariesProcessing*, *FormatsProcessing*, *NotationsProcessing*, *SignatureProcessing*, *ConstructorsProcessing*, *ClustersProcessing*, *IdentifyProcessing*, *ReductionProcessing*, *PropertiesProcessing*, *DefinitionsProcessing*, *EqualitiesProcessing*, *ExpansionsProcessing*, *TheoremsProcessing*, *SchemesProcessing*, *TheoremListsProcessing*, *SchemeListsProcessing*: *boolean*;
procedure *InitAccOptions*;
procedure *GetAccOptions*;

This code is used in section 56.

62. ⟨ Interface for *MakeEnv* command line options 62 ⟩ ≡
  { MakeEnv specific options: }
var *Accomodation*: *boolean* = *false*; *NewAccom*: *boolean* = *false*;
procedure *GetMEOptions*;

This code is used in section 56.

63. ⟨ Interface for transfer-specific command line options 63 ⟩ ≡
  { Transfer specific options: }
var *PublicLibr*: *boolean*;
procedure *GetTransfOptions*;

This code is used in section 56.

64. ⟨ Interface for other command line options 64 ⟩ ≡
{ Other options: }
var *CtrlCPressed*: *boolean* = *false*; *LongLines*: *boolean* = *false*; *QuietMode*: *boolean* = *false*;
StopOnError: *boolean* = *false*; *FinishingPass*: *boolean* = *false*; *ParserOnly*: *boolean* = *false*;
AnalyzerOnly: *boolean* = *false*; *CheckerOnly*: *boolean* = *false*; *SwitchOffUnifier*: *boolean* = *false*;
AxiomsAllowed: *boolean* = *false*;
procedure *GetOptions*;

This code is used in section 56.

65. The implementation begins by initializing the Accommodator specific options.

⟨ Implementation for mconsole.pas 65 ⟩ ≡
procedure *InitAccOptions*;
begin *InsertHiddenFiles* ← *true*; *VocabulariesProcessing* ← *true*; *FormatsProcessing* ← *true*;
NotationsProcessing ← *true*; *SignatureProcessing* ← *true*; *ConstructorsProcessing* ← *true*;
ClustersProcessing ← *true*; *IdentifyProcessing* ← *true*; *ReductionProcessing* ← *true*;
PropertiesProcessing ← *true*; *DefinitionsProcessing* ← *true*; *EqualitiesProcessing* ← *true*;
ExpansionsProcessing ← *true*; *TheoremsProcessing* ← *true*; *SchemesProcessing* ← *true*;
TheoremListsProcessing ← *false*; *SchemeListsProcessing* ← *false*;
end;

This code is used in section 54.

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

procedure *ResetAccOptions*;
begin *InsertHiddenFiles* ← *true*; *VocabulariesProcessing* ← *false*; *FormatsProcessing* ← *false*;
NotationsProcessing ← *false*; *SignatureProcessing* ← *false*; *ConstructorsProcessing* ← *false*;
ClustersProcessing ← *false*; *IdentifyProcessing* ← *false*; *ReductionProcessing* ← *false*;
PropertiesProcessing ← *false*; *DefinitionsProcessing* ← *false*; *EqualitiesProcessing* ← *false*;
ExpansionsProcessing ← *false*; *TheoremsProcessing* ← *false*; *SchemesProcessing* ← *false*;
TheoremListsProcessing ← *false*; *SchemeListsProcessing* ← *false*;
end;

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

- `-v` resets the accommodator options, and then toggles *VocabulariesProcessing* to true
- `-f`, `-p` resets the accommodator options, and then toggles *VocabulariesProcessing* to true (so far like `-v`), and then toggles *FormatsProcessing* to true.
- `-P` resets the accommodator options, and then toggles *VocabulariesProcessing* to true (so far like `-v`), and then toggles *FormatsProcessing* to true (so far like `-f` and `-p`), then toggles *TheoremListsProcessing* and *SchemeListsProcessing* to both be true.
- `-e` will do everything `-f` does, and then toggles *ConstructorsProcessing*, *SignatureProcessing*, *ClustersProcessing*, and *NotationsProcessing* to all be true.
- `-h` will set *InsertHiddenFalse* to false (presumably preventing Mizar from loading the "hidden" article, i.e., the primitive notions of `object`, `set`, `in`, `=`, and inequality will not be loaded).
- `-l` 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.

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

68. Similarly, we have *MakeEnv* specific options parsed from the command line flags.

procedure *GetMEOptions*;
  var i, j: *integer*;
  begin for $j \leftarrow 1$ **to** *ParamCount* **do**
    if *ParamStr*$(j)[1] =$ `'-'` **then**
      for $i \leftarrow 2$ **to** *length*(*ParamStr*(j)) **do**
        case *ParamStr*$(j)[i]$ **of**
        `'n'`: *NewAccom* ← *true*;
        `'a'`: *Accomodation* ← *true*;
        `'l'`: *LongLines* ← *true*;
        `'q'`: *QuietMode* ← *true*;
        `'s'`: *StopOnError* ← *true*;
        endcases;
  end;

69. The "other" options.

Notably, there is a feature to allow axioms, which is completely undocumented (and probably for good reason!). The axioms must appear in "`.axm`" files.

procedure *GetOptions*;
  var i, j: *integer*;
  begin for $j \leftarrow 1$ **to** *ParamCount* **do**
    if *ParamStr*$(j)[1] =$ `'-'` **then**
      for $i \leftarrow 2$ **to** *length*(*ParamStr*(j)) **do**
        case *ParamStr*$(j)[i]$ **of**
        `'q'`: *QuietMode* ← *true*;
        `'p'`: *ParserOnly* ← *true*;
        `'a'`: *AnalyzerOnly* ← *true*;
        `'c'`: *CheckerOnly* ← *true*;
        `'l'`: *LongLines* ← *true*;
        `'s'`: *StopOnError* ← *true*;
        `'u'`: *SwitchOffUnifier* ← *true*;
        `'x'`: *AxiomsAllowed* ← *true*;
        othercases *break*;
        endcases;
  if *ArticleExt* = `'.axm'` **then** *AxiomsAllowed* ← *true*;
  end;

70. Transfer specific options.

procedure *GetTransfOptions*;
  var *lOption*: *String*;
  begin *PublicLibr* ← *false*;
  if *ParamCount* ≥ 2 **then**
    begin *lOption* ← *ParamStr*(2);
    if (*length*(*lOption*) = 2) ∧ (*lOption*[1] ∈ [`'/'`, `'-'`]) **then** *PublicLibr* ← *UpCase*(*lOption*[2]) = `'P'`;
    end
  end;

71. We have a number of functions useful for "drawing", i.e., reporting progress and results (and so on).

```
var gComment: String = ´´;
  disable_io_checking;
procedure NoDisplayLine(fLine, fErrNbr : integer);
  begin end;
procedure InitDisplayLine(const aComment: String);
  begin gComment ← aComment; WriteLn; write(aComment); DisplayLine ← DisplayLineInCurPos
  end;
procedure DrawStr(const aStr: String);
  begin write(aStr) end;
procedure FinishDrawing;
  begin WriteLn;
  end;
procedure DrawTPass(const fPassName: String);
  begin write(fPassName) end;
procedure DrawMizarScreen(const aApplicationName: String);
  begin WriteLn(aApplicationName, ´,␣´, PCMizarVersionStr, ´␣(´, PlatformNameStr, ´)´);
  WriteLn(Copyright);
  end;
procedure Noise;
  begin
  if_not_def (WIN32)write(↑G↑G↑G);
  endif ;
  end ;
procedure EmptyParameterList;
  begin Noise; WriteLn; WriteLn(´****␣␣Empty␣Parameter␣List␣?␣****´); halt(2);
  end;
```

72. More such procedures, reporting the article processed, the time, etc.

```
procedure DrawArticleName(const fName: String);
  begin WriteLn(´Processing:␣´, fName); end;
procedure DrawPass(const aName: String);
  begin WriteLn; write(aName); end;
procedure DrawTime(const aTime: String);
  begin write(aTime); end;
procedure DrawVerifierExit(const aTime: String);
  begin WriteLn; WriteLn(´Time␣of␣mizaring:´, aTime);
  end;
```

73.

```
procedure DisplayLineInCurPos(fLine, fErrNbr : integer);
  begin if (¬CtrlCPressed) ∧ (¬QuietMode) then
    begin write(↑M + gComment + ´␣[´, fLine : 4);
    if fErrNbr > 0 then write(´␣*´, fErrNbr);
    write(´]´);
    end;
  if FinishingPass then
    begin write(´␣[´, fLine : 4);
    if fErrNbr > 0 then write(´␣*´, fErrNbr);
    write(´]´);
    end;
  end;
```

74.

```
procedure DrawMessage(const Msg1, Msg2: String);
  var Lh: byte;
  begin Noise; WriteLn; write(´****␣´, Msg1); Lh ← length(Msg1);
  if length(Msg2) > Lh then Lh ← length(Msg2);
  if Lh > length(Msg1) then write(´␣´ : Lh − length(Msg1));
  WriteLn(´␣****´);
  if Msg2 ≠ ´´ then
    begin write(´****␣´, Msg2);
    if Lh > length(Msg2) then write(´␣´ : Lh − length(Msg2));
    WriteLn(´␣****´);
    end;
  end;
```

75.

```
procedure BugInProcessor;
  begin DrawMessage(´Internal␣Error´, ´´); end;
procedure DrawIOResult(const FileName: String;
  I: byte);
  begin if I ∈ [2 . . 6, 12, 100] then
    begin if I = 12 then I ← 7
    else if I = 100 then I ← 8;
    DrawMessage(ErrMsg[I], ´Can´´t␣open␣´´␣´ + FileName + ´␣´´´)
    end
  else DrawMessage(´Can´´t␣open␣´´␣´ + FileName + ´␣´´´, ´´);
  halt(1);
  end;
```

76.

```
procedure DrawErrorsMSg(aErrorNbr : integer);
  begin if aErrorNbr > 0 then
    begin WriteLn;
    if aErrorNbr = 1 then WriteLn(´****␣1␣error␣detected´)
    else WriteLn(´****␣´, aErrorNbr, ´␣errors␣detected´);
    end;
  end;
```

File 3

PC Mizar Version

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

⟨ pcmizver.pas 77 ⟩ ≡
  ⟨ GNU License 4 ⟩

```
unit pcmizver;
  interface
  const PCMizarReleaseNbr = 8;
    PCMizarVersionNbr = 1;
    PCMizarVariantNbr = 14;
    CurrentYear = 2025;
    @{@&$IFDEF WIN32@}DirSeparator = ´\´;
    @{@&$ELSE@}DirSeparator = ´/´;
    @{@&$ENDIF@}
```

78. There are only four functions provided by this module.

```
function PCMizarVersionStr: String;
function VersionStr: String;
function PlatformNameStr: String;
function Copyright: String;
```

79. Their implementation is relativiely straightforward: just print the appropriate constants to the screen.

```
implementation
function Copyright: String;
  var s: String;
  begin Str(CurrentYear, s);
  Copyright ← ´Copyright␣(c)␣1990-´ + s + ´␣Association␣of␣Mizar␣Users´;
  end;
```

80.

```
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 ← ´0´ + lVar;
  @{@&$IFDEF VERALPHA@}lStr ← ´-alpha´;
  @{@&$ELSE@}lStr ← ´´;
  @{@&$ENDIF@}
  VersionStr ← lRel + ´.´ + lVer + ´.´ + lVar + lStr;
  end;
```

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

- `https://wiki.freepascal.org/Platform_defines`

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

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

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

function *PCMizarVersionStr*: *String*;
  begin *PCMizarVersionStr* ← ´Mizar␣Ver.␣´ + *VersionStr*;
  end;
end .

File 4

Mizar internal state

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

⟨ mstate.pas 83 ⟩ ≡
  ⟨ GNU License 4 ⟩
unit *mstate*;
  interface
    ⟨ Interface for `mstate.pas` 84 ⟩
  implementation
  uses *mizenv*, *pcmizver*, *monitor*, *errhan*, *mconsole*, *mtime*
  mdebug , *info* **end_mdebug** ;
  var *PassTime*: *longint*;
    ⟨ Implementation for `mstate.pas` 85 ⟩
  end

84. ⟨ Interface for `mstate.pas` 84 ⟩ ≡
procedure *InitPass*(**const** *aPassName*: *String*);
procedure *FinishPass*;
procedure *InitProcessing*(**const** *aProgName*, *aExt*: *String*);
procedure *ProcessingEnding*;
This code is used in section 83.

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

⟨ Implementation for `mstate.pas` 85 ⟩ ≡
procedure *InitPass*(**const** *aPassName*: *String*);
  begin *CurPos.Line* ← 1; *CurPos.Col* ← 1; *InitDisplayLine*(*aPassName*); *TimeMark*(*PassTime*);
  end;
procedure *FinishPass*;
  begin *FinishingPass* ← *true*;
  if *QuietMode* **then** *DisplayLine*(*CurPos.Line*, *ErrorNbr*);
  FinishingPass ← *false*; *DrawTime*(´␣␣´ + *ReportTime*(*PassTime*));
  end;
procedure *MError*(*Pos* : *Position*; *ErrNr* : *integer*);
  begin *WriteError*(*Pos*, *ErrNr*); *DisplayLine*(*CurPos.Line*, *ErrorNbr*);
  end;
This code is used in section 83.

86. We also have *MizarExitProc* as a private "helper" function.

var *_ExitProc*: *pointer*;
procedure *MizarExitProc*;
  begin *ExitProc* ← *_ExitProc*;
  disable_io_checking;
  if *IOResult* ≠ 0 **then** ;
  if ¬*StopOnError* **then** *DisplayLine*(*CurPos.Line*, *ErrorNbr*);
  PutError ← *WriteError*; *DrawVerifierExit*(*ReportTime*(*gStartTime*)); { Halt(ErrorCode); }
  enable_io_checking;
  end;

87.
procedure *InitProcessing*(**const** *aProgName*, *aExt*: *String*);
  begin *DrawMizarScreen*(*aProgName*);
  if *ParamCount* < 1 **then** *EmptyParameterList*;
  GetArticleName; *GetEnvironName*; *DrawArticleName*(*MizFileName* + *aExt*); *GetOptions*;
  InitExitProc; *FileExam*(*MizFileName* + *aExt*); *_ExitProc* ← *ExitProc*; *ExitProc* ← @*MizarExitProc*;
  PutError ← *MError*; *OpenErrors*(*MizFileName*);
  mdebug *OpenInfoFile*; **end_mdebug**
  end;

88. At the end, we should report the number of errors (if any were encountered).

procedure *ProcessingEnding*;
  begin if *ErrorNbr* > 0 **then**
    begin *DrawErrorsMsg*(*ErrorNbr*); *FinishDrawing*;
    end;
  end;

File 5

Monitor

89.

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

90. There are a few private helper functions in this module.

⟨ Implementation for `monitor.pas` 90 ⟩ ≡
procedure *_Halt_*(*ErrorCode* : *word*);
  begin *_IOResult* ← *IOResult*; *ErrorAddr* ← **nil**;
  if *ErrorCode* > 1 **then**
    case *ErrorCode* **of**
    2 . . 4: **begin** *ErrImm*(1000 + *ErrorCode*); *DrawMessage*(`´I/O␣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` 91 ⟩
      end;
    100 . . 101: **begin** *ErrImm*(1000 + *ErrorCode*); *DrawMessage*(`´I/O␣error´`, *ErrMsg*[*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␣overflow␣error´`, `´´`) **end**;
    203, 204: **begin** *ErrImm*(1000 + *ErrorCode*); *DrawMessage*(`´Heap␣overflow␣error´`, `´´`) **end**;
    208: **begin** *ErrImm*(1000 + *ErrorCode*); *DrawMessage*(`´Overlay␣manager␣not␣installed´`, `´´`) **end**;
    209: **begin** *ErrImm*(1000 + *ErrorCode*); *DrawMessage*(`´Overlay␣file␣read␣error´`, `´´`) **end**;
    210 . . 212: **begin** *ErrImm*(1000 + *ErrorCode*); *BugInProcessor* **end**;
    213: **begin** *ErrImm*(1000 + *ErrorCode*); *DrawMessage*(`´Collection␣Index␣out␣of␣range´`, `´´`) **end**;
    214: **begin** *ErrImm*(1000 + *ErrorCode*); *DrawMessage*(`´Collection␣overflow␣error´`, `´´`) **end**;
    215: **begin** *ErrImm*(1000 + *ErrorCode*); *DrawMessage*(`´Arithmetic␣overflow␣error´`, `´´`) **end**;
    216: **begin** *ErrImm*(1000 + *ErrorCode*); *DrawMessage*(`´General␣Protection␣fault´`, `´´`) **end**;
    217: **begin** *ErrImm*(1000 + *ErrorCode*); *DrawMessage*(`´Segmentation␣fault´`, `´´`) **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 *BugInProcessor*
      end;
    endcases;
  CloseErrors; *ExitProc* ← *_ExitProc*;
  if (*ErrorCode* = 0) ∧ (*ErrorNbr* ≠ 0) **then** *Halt*(1)
  else *Halt*(*ErrorCode*);
  end;

See also section 92.

This code is used in section 89.

91. ⟨ Handle runtime error cases for `monitor.pas` 91 ⟩ ≡

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:␣´` + *IntToStr*(*RTErrorCode*), `´´`)
else *BugInProcessor*
endcases;

This code is used in section 90.

92. The *MizExitProc* is a private "bail out" function.

⟨ Implementation for `monitor.pas` 90 ⟩ +≡

procedure *MizExitProc*;
begin
`@{@&$`*IFDEF IODEBUG* `@}` *ExitProc* ← *_ExitProc*;
`@{@&$`*ELSE* `@}` *_Halt_*(*ExitCode*);
`@{@&$`*ENDIF* `@}`
end;

93. We use the *MizExitProc* to initialize the *ExitProc* pointer.

procedure *InitExitProc*;
begin *ExitProc* ← @*MizExitProc*
end;

94. Initializing Control. This is a *heavily* system dependent piece of code. There are two ways to implement it (one way for Windows, another way for everyone else). Once we're done, we have to initialize the *_ExitProc* and invoke *InitCtrl*.

⟨ Non-windows FreePascal implemenation for *InitCtrl* 95 ⟩
⟨ Windows implemenation for *InitCtrl* 96 ⟩
begin *_ExitProc* ← *ExitProc*; *InitCtrl*;
end.

95. ⟨ Non-windows FreePascal implemenation for *InitCtrl* 95 ⟩ ≡

`@{@&$`*IFDEF FPC* `@}` `@{@&$`*IFNDEF WIN32* `@}`
procedure *CatchSignal*(*aSig* : *integer*); *cdecl*;
begin
case *aSig* **of**
SIGINT, *SIGQUIT*, *SIGTERM*: **begin** *CtrlCPressed* ← *true*; *RunTimeError*(1255); **end**;
endcases;
end;
var *NewSignal*, *OldSigInt*: *SignalHandler*;
procedure *InitCtrl*;
begin *NewSignal* ← *SignalHandler*(@*CatchSignal*); *OldSigInt* ← *fpSignal*(*SIGINT*, *NewSignal*);
OldSigInt ← *fpSignal*(*SIGQUIT*, *NewSignal*); *OldSigInt* ← *fpSignal*(*SIGTERM*, *NewSignal*);
end;
`@{@&$`*ENDIF* `@}@{@&$`*ENDIF* `@}`

This code is used in section 94.

96. Microsoft breaks everything. This is a mess because of them.

⟨ Windows implemenation for *InitCtrl* 96 ⟩ ≡
```
@{@&$IFDEF WIN32@}
  ⟨ Windows implemenation for CtrlSignal 99 ⟩
  @{@&$IFDEF FPC@}
    ⟨ FreePascal implementation of InitCtrl for Windows 97 ⟩
  @{@&$ENDIF@}
  @{@&$IFDEF DELPHI@}
    ⟨ Delphi implementation of InitCtrl for Windows 98 ⟩
  @{@&$ENDIF@}
@{@&$ENDIF@}
```
This code is used in section 94.

97. The FreePascal implementation is pretty succinct thanks to the libraries they provide.

⟨ FreePascal implementation of *InitCtrl* for Windows 97 ⟩ ≡
```
procedure InitCtrl;
  begin SetConsoleCtrlHandler(CtrlSignal, TRUE); end;
```
This code is used in section 96.

98. ⟨ Delphi implementation of *InitCtrl* for Windows 98 ⟩ ≡
```
procedure InitCtrl;
  var ConsoleMode, lConsoleMode: DWORD;
  begin if GetConsoleMode(GetStdHandle(STD_INPUT_HANDLE), ConsoleMode) then
    begin lConsoleMode ← ConsoleMode ∨ ENABLE_PROCESSED_INPUT;
        { Treat Ctrl+C as a signal }
    if SetConsoleMode(GetStdHandle(STD_INPUT_HANDLE), lConsoleMode) then
      begin SetConsoleCtrlHandler(@CtrlSignal, TRUE);
      end;
    end;
  end;
```
This code is used in section 96.

99. Windows requires a helper function *CtrlSignal* for this Microsoft mania.

⟨ Windows implemenation for *CtrlSignal* 99 ⟩ ≡
```
  ⟨ FreePascal declaration of CtrlSignal for Windows 100 ⟩
  ⟨ Delphi declaration of CtrlSignal for Windows 101 ⟩
  begin   { TRUE: do not call next handler in the queue, FALSE: call it }
  CtrlCPressed ← true; RunTimeError(1255); CtrlSignal ← true;   { ExitProcess(1); }
  end;
```
This code is used in section 96.

100. ⟨ FreePascal declaration of *CtrlSignal* for Windows 100 ⟩ ≡
```
  @{@&$IFDEF FPC@}
function CtrlSignal(aSignal : DWORD): WINBOOL; stdcall;
    @{@&$ENDIF@}
```
This code is used in section 99.

101. ⟨ Delphi declaration of *CtrlSignal* for Windows 101 ⟩ ≡
```
  @{@&$IFDEF DELPHI@}
function CtrlSignal(aSignal : DWORD): BOOL; cdecl;
  @{@&$ENDIF@}
```
This code is used in section 99.

File 6

Error handling

102.

⟨errhan.pas 102⟩ ≡
  ⟨GNU License 4⟩
unit *errhan*;
  interface
    ⟨Interface for `errhan.pas` 103⟩
  implementation
  uses *mconsole*, *mizenv*;
    ⟨Implementation for `errhan.pas` 106⟩
  end ;

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

⟨Interface for `errhan.pas` 103⟩ ≡
type *Position* = ⟨Declare *Position* as **record** 104⟩;
  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 section 105.

This code is used in section 102.

104. Position is just a pair of integers recording the line and offset ("column").

⟨Declare *Position* as **record** 104⟩ ≡
  record *Line*, *Col*: *integer*
  end

This code is used in section 103.

105. And we just provide the public-facing functions and procedures.

⟨ Interface for `errhan.pas` 103 ⟩ +≡
procedure *Error*(*Pos* : *Position*; *ErrNr* : *integer*);
procedure *ErrImm*(*ErrNr* : *integer*);
procedure *WriteError*(*Pos* : *Position*; *ErrNr* : *integer*);
procedure *OpenErrors*(*FileName* : *String*);
procedure *AppendErrors*(*FileName* : *String*);
procedure *EraseErrors*;
procedure *CloseErrors*;
procedure *OverflowError*(*ErrorCode* : *word*);
procedure *Mizassert*(*ErrorCode* : *word*; *Cond* : *boolean*);
procedure *RunTimeError*(*ErrorCode* : *word*);

106. 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.

⟨ Implementation for `errhan.pas` 106 ⟩ ≡
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;

This code is used in section 102.

107. We also can write errors to a file. This requires keeping track of the file (dubbed *Errors*) and whether it has been opened or not (in the Boolean condition *OpenedErrors*).

Note this takes advantage of **with** to destructure *Pos* into a *Line* and *Col* for us.

var *Errors*: *text*; { file name for errors file }
  OpenedErrors: *boolean* = *false*; { have we opened it yet? }
procedure *WriteError*(*Pos* : *Position*; *ErrNr* : *integer*);
  begin if ¬*OpenedErrors* **then** *RunTimeError*(2001);
  with *Pos* **do** *WriteLn*(*Errors*, *Line*, ´␣´, *Col*, ´␣´, *ErrNr*);
  end;

108. Opening an errors file. We can open an errors file, which will reset the *ErrorNbr* counter to zero and re-initialize *CurPos* to line 1 and column 1.

When *PutError* is **nil**, we initialize it to be *WriteError*.

```
procedure OpenErrors(FileName : String);
  begin if ExtractFileExt(FileName) = ´´ then FileName ← FileName + ´.err´;
  Assign(Errors, FileName); without_io_checking(Rewrite(Errors));
  if IOResult ≠ 0 then
    begin DrawMessage(´Can´´t␣open␣errors␣file␣´´´ + FileName + ´´´␣for␣writing´, ´´); halt(1);
    end;
  OpenedErrors ← true; ErrorNbr ← 0;
  with CurPos do
    begin Line ← 1; Col ← 1
    end;
  if @PutError = nil then PutError ← WriteError;
  end;
```

109. Appending errors to the errors file.

```
procedure AppendErrors(FileName : String);
  begin OpenedErrors ← true;
  if ExtractFileExt(FileName) = ´´ then FileName ← FileName + ´.err´;
  Assign(Errors, FileName); ErrorNbr ← 0;
  with CurPos do
    begin Line ← 1; Col ← 1
    end;
  without_io_checking(append(Errors));
  if ioresult ≠ 0 then Rewrite(Errors);
  end;
```

110. We can also close the errors file and unset the *Errors* variable, "forgetting" where we logged the errors.

```
procedure EraseErrors;
  begin if OpenedErrors then
    begin OpenedErrors ← false; close(Errors); erase(Errors);
    end;
  end;
```

111. We can also just close the errors file.

```
procedure CloseErrors;
  begin if OpenedErrors then
    begin OpenedErrors ← false; close(Errors);
    end;
  end;
```

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

```
procedure OverflowError(ErrorCode : word);
  begin RTErrorCode ← ErrorCode; OverflowErroor ← true; RunError(97);
  end;
```

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

procedure *MizAssert*(*ErrorCode* : *word*; *Cond* : *boolean*);
  begin if ¬*Cond* **then**
    begin *RTErrorCode* ← *ErrorCode*; *RunError*(98);
    end;
  end;

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

procedure *RunTimeError*(*ErrorCode* : *word*);
  begin *RTErrorCode* ← *ErrorCode*; *RunError*(99);
  end;

File 7

Time utilities

115. This is another heavily "system dependent" library.

⟨ mtime.pas 115 ⟩ ≡
  ⟨ GNU License 4 ⟩
unit *mtime*;
  interface
    ⟨ Interface for `mtime.pas` 116 ⟩
  implementation
    ⟨ Implementation for `mtime.pas` 118 ⟩
  end ;

116. ⟨ Interface for `mtime.pas` 116 ⟩ ≡
procedure *TimeMark*(**var** W : *longint*);
function *ElapsedTime*(W : *longint*): *longint*;
procedure *MUnpackTime*(W : *longint*; **var** H, M, S, F : *word*);
function *ReportTime*(W : *longint*): *String*;

This code is used in section 115.

117. We also have one global variable tracking the start time.

var *gStartTime*: *longint*;

118. The implementation begins with a rather *thorny* digression depending on which compiler we're using.

⟨ Implementation for `mtime.pas` 118 ⟩ ≡
  ⟨ Timing utilities **uses** for Delphi 119 ⟩
  ⟨ Timing utilities **uses** for FreePascal 120 ⟩

See also section 121.

This code is used in section 115.

119. Delphi simply requires us to introduce a constant for milliseconds.

⟨ Timing utilities **uses** for Delphi 119 ⟩ ≡
  `@{@&$`*IFDEF DELPHI* `@}`
  uses *windows*;
const $cmSecs = 1000$;
  `@{@&$`*ENDIF* `@}`

This code is used in section 118.

120. FreePascal requires a bit more work, alas.

⟨ Timing utilities **uses** for FreePascal 120 ⟩ ≡

```
  @{@&$IFDEF FPC@}
  uses dos;
const cmSecs = 100;
type TSystemTime =
  record wHour: word;
  wMinute: word;
  wSecond: word;
  wMilliseconds: word;
  end;
procedure GetLocalTime(var aTime : TSystemTime);
  begin with aTime do GetTime(wHour, wMinute, wSecond, wMilliseconds);
  end;
  @{@&$ENDIF@}
```

This code is used in section 118.

121. Now we can happily plug along implementing the functions we need.

⟨ Implementation for `mtime.pas` 118 ⟩ +≡

```
function SystemTimeToMiliSec(const fTime: TSystemTime): longint;
  begin SystemTimeToMiliSec ← fTime.wHour ∗ (3600 ∗ cmSecs) + fTime.wMinute ∗ longint(60 ∗
      cmSecs) + fTime.wSecond ∗ cmSecs + fTime.wMilliseconds;
  end;
```

122. We "start the clock".

```
procedure TimeMark(var W : longint);
  var SystemTime: TSystemTime;
  begin GetLocalTime(SystemTime); W ← SystemTimeToMiliSec(SystemTime);
  end;
```

123. We measure the time lapse since we "started the clock".

```
function ElapsedTime(W : longint): longint;
  var T: longint; SystemTime: TSystemTime;
  begin GetLocalTime(SystemTime); T ← SystemTimeToMiliSec(SystemTime) − W;
  if T < 0 then T ← 86400 ∗ cmSecs + T;
  ElapsedTime ← T;
  end;
```

124. We can transform an interval of time (in milliseconds) into hours, minutes, seconds, a fractional amount of time.

```
procedure MUnpackTime(W : longint; var H, M, S, F : word);
  begin H ← W div (3600 ∗ cmSecs); M ← (W − H ∗ 3600 ∗ cmSecs) div (60 ∗ cmSecs);
  S ← (W − H ∗ 3600 ∗ cmSecs − M ∗ 60 ∗ cmSecs) div cmSecs;
  F ← W − H ∗ 3600 ∗ cmSecs − M ∗ 60 ∗ cmSecs − S ∗ cmSecs;
  end;
```

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

function *LeadingZero*(w : *word*): *String*;
  var *lStr*: *String*;
  begin *Str*(w : 0, *lStr*);
  if *length*(*lStr*) = 1 **then** *lStr* ← ´0´ + *lStr*;
  LeadingZero ← *lStr*;
  end;

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

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

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

  begin *TimeMark*(*gStartTime*);
  end.

File 8

Arbitrary precision arithmetic

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

Note:

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

⟨ numbers.pas 128 ⟩ ≡
  ⟨ GNU License 4 ⟩
unit *numbers*;
  interface
    ⟨ Basic arithmetic operations declarations 129 ⟩
    ⟨ Types for arbitrary-precision arithmetic 130 ⟩
    ⟨ Zero and units for arbitrary-precision 131 ⟩
    ⟨ Rational arithmetic declarations 132 ⟩
    ⟨ Predicate declarations for arbitrary-precision arithmetic 133 ⟩
    ⟨ Declare public complex-valued arbitrary precision arithmetic 134 ⟩
    ⟨ Declare public comparison operators for arbitrary-precision numbers 135 ⟩
  implementation
  uses *mizenv*
    @{@&$*IFDEF CH_REPORT*@} , *req_info*, *prephan*, *builtin*@{@&$*ENDIF*@}
      mdebug , *info* **end_mdebug**;

  ⟨ Trim leading zeros from arbitrary-precision integers 136 ⟩
  ⟨ Check if arbitrary-precision integers are zero 137 ⟩
  ⟨ Absolute value for an arbitrary-precision number 138 ⟩
  ⟨ Test if one arbitrary-precision number is less than or equal to another 139 ⟩
  ⟨ Arithmetic for arbitrary-precision integers 143 ⟩
  ⟨ Arbitrary-precision rational arithmetic 160 ⟩
  ⟨ Complex-rational arbitrary-precision arithmetic 169 ⟩

129. ⟨ Basic arithmetic operations declarations 129 ⟩ ≡
function *Add*(*a*, *b* : *String*): *String*;
function *Sub*(*a*, *b* : *String*): *String*;
function *Mul*(*a*, *b* : *String*): *String*;
function *Diva*(*a*, *b* : *String*): *String*;
{ *Note: divides absolute values and preserves the sign of the division }
function *_Div*(*a*, *b* : *String*): *String*;
function *_Mod*(*a*, *b* : *String*): *String*;
function *GCD*(*a*, *b* : *String*): *String*; { *Note: always returns a positive value }
function *LCM*(*a*, *b* : *String*): *String*; { *Note: always returns a positive value }
function *Abs*(*a* : *String*): *String*;
function *IsPrime*(*a* : *String*): *boolean*;
function *Divides*(*a*, *b* : *String*): *boolean*;

This code is used in section 128.

130. Rational numbers are a pair of arbitrary precision integers (represented as a String).

Rational complex numbers are represented by a pair of rational numbers in Cartesian form $z = p + \mathrm{i}q$.

⟨ Types for arbitrary-precision arithmetic 130 ⟩ ≡
type *Rational* = **record** *Num*, *Den*: *String*
end;
RComplex = **record** *Re*, *Im*: *Rational*
end;

This code is used in section 128.

131. ⟨ Zero and units for arbitrary-precision 131 ⟩ ≡
const *RZero*: *Rational* = (*Num* : ´0´; *Den* : ´1´);
ROne: *Rational* = (*Num* : ´1´; *Den* : ´1´);
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´));

This code is used in section 128.

132. ⟨ Rational arithmetic declarations 132 ⟩ ≡
procedure *RationalReduce*(**var** *r* : *Rational*);
function *RationalAdd*(**const** *r1*, *r2*: *Rational*): *Rational*;
function *RationalSub*(**const** *r1*, *r2*: *Rational*): *Rational*;
function *RationalNeg*(**const** *r1*: *Rational*): *Rational*;
function *RationalMult*(**const** *r1*, *r2*: *Rational*): *Rational*;
function *RationalInv*(**const** *r*: *Rational*): *Rational*;
function *RationalDiv*(**const** *r1*, *r2*: *Rational*): *Rational*;
function *RationalEq*(**const** *r1*, *r2*: *Rational*): *boolean*;
function *RationalLE*(**const** *r1*, *r2*: *Rational*): *boolean*;
function *RationalGT*(**const** *r1*, *r2*: *Rational*): *boolean*;

This code is used in section 128.

133. ⟨ Predicate declarations for arbitrary-precision arithmetic 133 ⟩ ≡
function *IsintegerNumber*(**const** *z*: *RComplex*): *boolean*;
function *IsNaturalNumber*(**const** *z*: *RComplex*): *boolean*;
function *IsPrimeNumber*(**const** *z*: *RComplex*): *boolean*;
function *AreEqComplex*(**const** *z1*, *z2*: *RComplex*): *boolean*;
function *IsEqWithInt*(**const** *z*: *RComplex*;
n: *longint*): *boolean*;
function *IsRationalLE*(**const** *z1*, *z2*: *RComplex*): *boolean*;
function *IsRationalGT*(**const** *z1*, *z2*: *RComplex*): *boolean*;

This code is used in section 128.

134. ⟨ Declare public complex-valued arbitrary precision arithmetic 134 ⟩ ≡
function *IntToComplex*(*x* : *integer*): *RComplex*;
function *ComplexAdd*(**const** *z1*, *z2*: *RComplex*): *RComplex*;
function *ComplexSub*(**const** *z1*, *z2*: *RComplex*): *RComplex*;
function *ComplexNeg*(**const** *z*: *RComplex*): *RComplex*;
function *ComplexMult*(**const** *z1*, *z2*: *RComplex*): *RComplex*;
function *ComplexInv*(**const** *z*: *RComplex*): *RComplex*;
function *ComplexDiv*(**const** *z1*, *z2*: *RComplex*): *RComplex*;
function *ComplexNorm*(**const** *z*: *RComplex*): *Rational*;

This code is used in section 128.

135. ⟨ Declare public comparison operators for arbitrary-precision numbers 135 ⟩ ≡
function *CompareInt*(*X1*, *X2* : *Longint*): *integer*;
function *CompareIntStr*(*X1*, *X2* : *String*): *integer*;
function *CompareComplex*(**const** *z1*, *z2*: *RComplex*): *integer*;

This code is used in section 128.

136. If we are given single character String consisting of zero or the empty String, then we are done.

If we are given anything else, we find the first index (from the left) of a nonzero character. Then we create a copy of the subString starting from the first nonzero digit to the rest of the String.

⟨ Trim leading zeros from arbitrary-precision integers 136 ⟩ ≡

```
function trimlz(a : String): String;
  var i: integer;
  begin if (a = ´0´) ∨ (a = ´´) then trimlz ← a
  else begin i ← 0;
    repeat i ← i + 1;
      if a[i] ≠ ´0´ then break;
    until i = length(a);
    trimlz ← copy(a, i, length(a));
    end;
  end;
```

This code is used in section 128.

137. First, we check if a starts with "−0". If so, replace a with 0. Then we do the same thing with b.

We invoke *trimlz* on a and store the result in *a1*. If $a1 \neq a$, then we update $a \leftarrow a1$.

Then we do likewise on b.

⟨ Check if arbitrary-precision integers are zero 137 ⟩ ≡

```
procedure checkzero(var a, b : String);
  var a1, b1: String;
  begin if copy(a, 1, 2) = ´-0´ then
    begin
    @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, ´a=-0´);
    @{@&$ENDIF @}a ← ´0´;
    end;
  if copy(b, 1, 2) = ´-0´ then
    begin
    @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, ´b=-0´);
    @{@&$ENDIF @}
    b ← ´0´;
    end;
  a1 ← trimlz(a);
  if a1 ≠ a then
    begin
    @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, ´ZEROS1:´, a);
    @{@&$ENDIF @}a ← a1;
    end;
  b1 ← trimlz(b);
  if b1 ≠ b then
    begin
    @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, ´ZEROS2:´, b);
    @{@&$ENDIF @}b ← b1;
    end;
  end;
```

This code is used in section 128.

138. Since arbitrary precision numbers (as Strings) are negative if they begin with a leading "-" character, it is easy to obtain the absolute value (just delete the minus sign).

⟨ Absolute value for an arbitrary-precision number 138 ⟩ ≡

```
function Abs(a : String): String;
  begin if length(a) > 0 then
    if a[1] = ´-´ then delete(a, 1, 1);
  Abs ← a;
  end;
```

This code is used in section 128.

139. When checking $a \le b$ 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.

⟨ Test if one arbitrary-precision number is less than or equal to another 139 ⟩ ≡

```
function _leq(a, b : String): boolean;
  var i, x, y, z: integer;
  begin @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, ´_leq(´, a, ´,´, b, ´)´);
  @{@&$ENDIF @} checkzero(a, b);
  if length(a) < length(b) then _leq ← true
  else if length(a) > length(b) then _leq ← false
    else begin for i ← 1 to length(a) do
        begin val(a[i], x, z); val(b[i], y, z);
        if x > y then
          begin _leq ← false; exit;
          end;
        if x < y then
          begin _leq ← true; exit;
          end;
        end;
      _leq ← true;
      end;
  end;
```

This code is used in section 128.

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

When a does not start with a minus sign, but b *does* start with a minus sign, then we're done: $b < a$.

When neither a nor b starts with a minus sign, then we use *_leq(a,b)* to determine the result.

```
function leq(a, b : String): boolean;
  begin @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, ´leq(´, a, ´,´, b, ´)´);
  @{@&$ENDIF @} checkzero(a, b);
  if a = b then leq ← true
  else begin if (a[1] = ´-´) ∧ (b[1] ≠ ´-´) then leq ← true;
    if (a[1] = ´-´) ∧ (b[1] = ´-´) then leq ← ¬_leq(abs(a), abs(b));
    if (a[1] ≠ ´-´) ∧ (b[1] = ´-´) then leq ← false;
    if (a[1] ≠ ´-´) ∧ (b[1] ≠ ´-´) then leq ← _leq(a, b);
    end;
  end;
```

141. Testing if $a \ge b$ is simply testing if $b \le a$ after normalizing the Strings.

```
function geq(a, b : String): boolean;
  begin @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, ´geq(´, a, ´,´, b, ´)´);
  @{@&$ENDIF @} checkzero(a, b);
  geq ← (¬leq(a, b)) ∨ (a = b);
  end;
```

142. Similarly, we may check if $a < b$ by testing $a \neq b$ and $a \leq b$.

function *le*(a, b : *String*): *boolean*;
  begin `@{@&$`*IFDEF DEBUGNUM* `@}` *WriteLn*(*infofile*, ´`le(`´, a, ´`,`´, b, ´`)`´);
  `@{@&$`*ENDIF* `@}`*checkzero*(a, b); $le \leftarrow (a \neq b) \land (leq(a, b))$;
  end;

function *gt*(a, b : *String*): *boolean*;
  begin `@{@&$`*IFDEF DEBUGNUM* `@}` *WriteLn*(*infofile*, ´`gt(`´, a, ´`,`´, b, ´`)`´);
  `@{@&$`*ENDIF* `@}`*checkzero*(a, b); $gt \leftarrow \neg leq(a, b)$;
  end;

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

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

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

$$\begin{array}{r} a_n \ldots a_{m+1}\, a_m\, a_{m-1} \ldots a_1 \\ +\qquad\qquad b_m\, b_{m-1} \ldots b_1 \\ \hline \end{array} \tag{143.1}$$

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

$$\begin{array}{r} a_m\, a_{m-1} \ldots a_1 \\ +\; b_m\, b_{m-1} \ldots b_1 \\ \hline c_{m+1}\, r_m\, r_{m-1} \cdots r_1 \end{array} \tag{143.2}$$

Then we will compute

$$\begin{array}{r} a_n \ldots a_{m+1} \\ +\qquad c_{m+1} \\ \hline r_{n+1}\, r_n\, \ldots\, r_{m+1} \end{array} \tag{143.3}$$

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

```
⟨ Arithmetic for arbitrary-precision integers 143 ⟩ ≡
function _Add(a, b : String): String;
  var c, x, y, z, v: integer; i: integer; a1, b1, s, r: String;
  begin a1 ← a; b1 ← b;
  @{@&$IFDEF DEBUGNUM @}WriteLn(infofile, ´_Add(´, a1, ´,´, b1, ´)´);
  @{@&$ENDIF @}checkzero(a1, b1);
  if length(a1) < length(b1) then
    begin s ← b1; b1 ← a1; a1 ← s; end;
  r ← ´´; c ← 0;
  begin for i ← 0 to length(b1) − 1 do   { step 1, Eq (143.2) }
    begin val(a1[length(a1) − i], x, z); val(b1[length(b1) − i], y, z);
    if x + y + c > 9 then
      begin v ← (x + y + c) − 10; c ← 1;
      end
    else begin v ← x + y + c; c ← 0;
      end;
    Str(v, s); r ← s + r;
    end;
  for i ← length(b1) to length(a1) − 1 do   { step 2, Eq (143.3) }
    begin val(a1[length(a1) − i], x, z);
    if x + c > 9 then
      begin v ← (x + c) − 10; c ← 1;
      end
    else begin v ← x + c; c ← 0;
      end;
    Str(v, s); r ← s + r;
    end;
  if c = 1 then r ← ´1´ + r;
  end;
  _Add ← trimlz(r);
  end;
```

See also sections 146 and 151.

This code is used in section 128.

144. Subtraction is a bit trickier, because of the "borrowing" operation.

Also note that *_Sub(a,b)* will start by computing $a_1 \leftarrow \max(a,b)$ and $b_1 \leftarrow \min(a,b)$, then return $a_1 - b_1$.

```
function _Sub(a, b : String): String;
  var x, y, z, v: integer; i: integer; a1, b1, s, r: String;
    ⟨ Borrow 1 for Sub_ 145 ⟩
    begin a1 ← a; b1 ← b;
    @{@&$IFDEF DEBUGNUM @}WriteLn(infofile, '_Sub(', a1, ',', b1, ')');
    @{@&$ENDIF @}checkzero(a1, b1);
    if ¬_leq(b1, a1) then begin s ← b1; b1 ← a1; a1 ← s; end;
    r ← '';
      begin
        for i ← 0 to length(b1) − 1 do
          begin val(a1[length(a1) − i], x, z); val(b1[length(b1) − i], y, z);
          if x < y then
            begin borrow(length(a1) − i); x ← x + 10; end;
          v ← x − y; Str(v, s); r ← s + r;
          end;
        for i ← length(a1) − length(b1) downto 1 do
          begin r ← a1[i] + r; end;
      end;
    _Sub ← trimlz(r);
  end ;
```

145. This is a private "helper function" for subtraction.

```
⟨ Borrow 1 for Sub_ 145 ⟩ ≡
procedure Borrow(k : integer);
  var xx, zz: integer; sx: String;
  begin val(a1[k − 1], xx, zz);
  if xx ≥ 1 then
    begin xx ← xx − 1; Str(xx, sx); a1[k − 1] ← sx[1];
    end
  else begin a1[k − 1] ← '9'; borrow(k − 1);
    end;
  end;
```

This code is used in section 144.

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

```
⟨ Arithmetic for arbitrary-precision integers 143 ⟩ +≡
function _Mul1(a : String; y : integer): String;
  var c, x, z, v: integer; i: integer; s, r: String;
  begin
    @{@&$IFDEF DEBUGNUM @}WriteLn(infofile, ´_Mul1(´, a, ´,´, y, ´)´);
    @{@&$ENDIF @}r ← ´´; c ← 0;
    begin
      for i ← 0 to length(a) − 1 do
        begin val(a[length(a) − i], x, z);
        if x ∗ y + c > 9 then
          begin v ← (x ∗ y + c) mod 10; c ← (x ∗ y + c) div 10; end
        else begin v ← x ∗ y + c; c ← 0; end;
        Str(v, s); r ← s + r;
        end;
      if c ≠ 0 then
        begin Str(c, s); r ← s + r; end;

    end;
    _mul1 ← trimlz(r);
  end;
```

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

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

148. Division. The basic design is similar to multiplication. We will try to divide a by $b \times 10^k$ (which is zero whenever $b \times 10^k > a$).

```
function _Div1(a, b : String): String;
  var i: integer; r: String;
  begin
  @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, ´_Div1(´, a, ´,´, b, ´)´);
  @{@&$ENDIF @} checkzero(a, b);
  if ¬_leq(b, a) then _div1 ← ´0´
  else for i ← 9 downto 1 do
      begin Str(i, r);
      if _leq(_mul(b, r), a) then
        begin _div1 ← trimlz(r); exit; end;
      end;
  end;
```

149.

```
function _Div(a, b : String): String;
  var z, c, i: integer; s, r, rs: String; b_GPC: boolean;
    ⟨ Get the next digit for dividing arbitrary-precision integers 150 ⟩
  begin
    @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, ´_Div(´, a, ´,´, b, ´)´);
    @{@&$ENDIF @} checkzero(a, b);
    if a = b then _div ← ´1´
    else if ¬_leq(b, a) then _div ← ´0´
    else begin s ← ´´; r ← ´´; z ← 1;
      for i ← 1 to length(b) do s ← s + a[i];
      if ¬_leq(b, s) then
        begin s ← s + a[length(b) + 1]; z ← length(b) + 1; end
      else begin z ← length(b); end;
      repeat rs ← _div1(s, b); r ← r + rs; gets; b_GPC ← _leq(b, s);
      until ¬b_GPC;
      _div ← trimlz(r);
    end;

  end;
```

150. ⟨ Get the next digit for dividing arbitrary-precision integers 150 ⟩ ≡

```
procedure gets;
  var j: integer;
  begin c ← 1; s ← _Sub(s, _mul(rs, b));
  if (s = ´0´) ∧ (trimlz(copy(a, z + c, length(a))) = ´0´) then
    begin @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, ´Rewriting␣zeros:´, copy(a, z + c, length(a)));
    @{@&$ENDIF @}r ← r + copy(a, z + c, length(a)); exit;
    end;
  if z + 1 ≤ length(a) then
    begin s ← s + a[z + 1]; inc(c);
    if (¬_leq(b, s)) then r ← r + ´0´;
    end;
  while (¬_leq(b, s)) ∧ (z + c ≤ length(a)) do
    begin s ← s + a[z + c]; inc(c);
    if (¬_leq(b, s)) then r ← r + ´0´;
    end;
  z ← z + c − 1;
  end; { gets }
```

This code is used in section 149.

151. Modulo. We can compute a **mod** b by observing if $a < b$ then we should obtain a. Otherwise, we should compute $r \leftarrow a$ **div** b, then $a - rb$ is a **mod** b.

⟨ Arithmetic for arbitrary-precision integers 143 ⟩ +≡

```
function _Mod(a, b : String): String;
  var r: String;
  begin @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, ´_Mod(´, a, ´,´, b, ´)´);
  @{@&$ENDIF @} checkzero(a, b);
  if le(a, b) then r ← a
  else r ← _Sub(a, _mul(b, _div (a, b)));
  _Mod ← trimlz(r);
  @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, ´End␣_Mod:´, r);
  @{@&$ENDIF @}
  end;
```

152. Greatest common divisor. We can compute $\gcd(a,b)$ first by setting $a_1 \leftarrow |a|$ and $b_1 \leftarrow |b|$ (since $\gcd(a,b) = \gcd(|a|,|b|)$). Then we handle the special cases:
(1) $a_1 = 1$ or $b_1 = 1$, then $\gcd(a_1,b_1) = 1$
(2) $a_1 = 0$ and $b_1 \neq 0$, then $\gcd(a_1,b_1) = b_1$
(3) $a_1 \neq 0$ and $b_1 = 0$, then $\gcd(a_1,b_1) = a_1$
(4) $a_1 = b_1$, then $\gcd(a_1,b_1) = a_1$

Otherwise, we end up in the default case, which is handled by the **while** loop.

```
function GCD(a, b : String): String;
  label ex;
  var a1, b1, p, r: String;
  begin a1 ← a; b1 ← b;
  @{@&$IFDEF DEBUGNUM @}WriteLn(infofile, ´GCD(´, a1, ´,´, b1, ´)´);
  @{@&$ENDIF @}checkzero(a1, b1); a1 ← abs(a1); b1 ← abs(b1);
  if (a1 = ´1´) ∨ (b1 = ´1´) then
    begin r ← ´1´; goto ex; end;
  if (a1 = ´0´) ∧ (b1 ≠ ´0´) then
    begin r ← b1; goto ex; end;
  if (b1 = ´0´) ∧ (a1 ≠ ´0´) then
    begin r ← a1; goto ex; end;
  if a1 = b1 then
    begin GCD ← a1; r ← a1; goto ex; end;
  while gt(b1, ´0´) do
    begin p ← b1; b1 ← _mod (a1, b1); a1 ← p end;
  r ← a1;
ex: GCD ← r;
  @{@&$IFDEF DEBUGNUM @}WriteLn(infofile, ´End␣GCD:´, r);
  @{@&$ENDIF @}
  end;
```

153. Least common multiple. We recall $\operatorname{lcm}(a,b) = |ab|/\gcd(|a|,|b|)$.

```
function LCM(a, b : String): String;
  var a1, b1, r: String;
  begin a1 ← a; b1 ← b;
  @{@&$IFDEF DEBUGNUM @}WriteLn(infofile, ´LCM(´, a1, ´,´, b1, ´)´);
  @{@&$ENDIF @}checkzero(a1, b1); a1 ← abs(a1); b1 ← abs(b1);
  r ← Diva(Mul(a1, b1), GCD(a1, b1)); LCM ← r;
  @{@&$IFDEF DEBUGNUM @}WriteLn(infofile, ´End␣LCM:´, r);
  @{@&$ENDIF @}
  end;
```

154. Addition. This is a bit obfuscated with the reliance of **goto** *ex*, but the basic idea is (recalling that *_Sub(a,b)* calculates $\max(a,b) - \min(a,b)$ for $a \geq 0$ and $b \geq 0$):

(1) If $a < 0$ and $b < 0$, then $a + b = -(|a| + |b|)$
(2) Else if $a \geq 0$ and $b \geq 0$, then $a + b$ is computed using *_Add*
(3) Else if $a < 0$ and $b \geq 0$, then we have two cases
  (i) If $|a| \geq b$, compute $a + b = -(|a| - b)$
  (ii) Otherwise, $a + b = b - |a|$
(4) Else if $a \geq 0$ and $b < 0$, then $a + b = a - |b|$
(5) Otherwise, when $a \geq 0$ and $b \geq 0$, $a + b$ is computed using *_Add*.

```
function Add(a, b : String): String;
  label ex;
  var r: String;
  begin @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, 'Add(', a, ',', b, ')');
  @{@&$ENDIF @} checkzero(a, b);
  if (a[1] = '-') ∧ (b[1] = '-') then
    begin r ← '-' + _Add(abs(a), abs(b));
    if r = '-0' then r ← '0';
    goto ex;
    end;
  if (a[1] ≠ '-') ∧ (b[1] ≠ '-') then
    begin r ← _Add(a, b); goto ex; end;
  if (a[1] = '-') ∧ (b[1] ≠ '-') then
    if gt(abs(a), b) then
      begin r ← '-' + _Sub(abs(a), b);
      if r = '-0' then r ← '0';
      goto ex;
      end
    else begin r ← _Sub(abs(a), b); goto ex; end;
  if (a[1] ≠ '-') ∧ (b[1] = '-') then
    if gt(abs(b), a) then
      begin r ← '-' + _Sub(abs(b), a);
      if r = '-0' then r ← '0';
      goto ex;
      end
    else begin r ← _Sub(abs(b), a); goto ex; end;
ex: Add ← r;
  @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, 'End␣Add:', r);
  @{@&$ENDIF @}
  end;
```

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

(1) If $a < 0$ and $b \geq 0$, then $a - b = -(|a| + b)$
(2) Else if $a \geq 0$ 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 $|a| \leq |b|$, so $a - b = |a| - |b|$
(4) Else if $a \geq 0$ and $b \geq 0$, then we have two cases
  (i) If $b > a$, then $a - b = -(b - a)$
  (ii) Otherwise compute $a - b$ using *_Sub(a,b)*

Testing if $x < 0$ is done by checking $\operatorname{sgn}(x) = -1$, and $x \geq 0$ tests if $\operatorname{sgn}(x) \neq -1$.

```
function Sub(a, b : String): String;
  label ex;
  var r: String;
  begin @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, 'Sub(', a, ',', b, ')');
  @{@&$ENDIF @} checkzero(a, b);
  if (a[1] = '-') ∧ (b[1] ≠ '-') then
    begin r ← '-' + _Add(abs(a), b);
    if r = '-0' then r ← '0';
    goto ex;
    end;
  if (a[1] ≠ '-') ∧ (b[1] = '-') then
    begin r ← _Add(a, abs(b)); goto ex; end;
  if (a[1] = '-') ∧ (b[1] = '-') then
    if gt(abs(a), abs(b)) then
      begin r ← '-' + _Sub(abs(a), abs(b));
      if r = '-0' then r ← '0';
      goto ex;
      end
    else begin r ← _Sub(abs(a), abs(b)); goto ex; end;
  if (a[1] ≠ '-') ∧ (b[1] ≠ '-') then
    if gt(b, a) then
      begin r ← '-' + _Sub(b, a);
      if r = '-0' then r ← '0';
      goto ex;
      end
    else begin r ← _Sub(a, b); goto ex; end;
ex: Sub ← r;
  @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, 'End␣Sub:', r);
  @{@&$ENDIF @}
  end;
```

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

```
function Mul(a, b : String): String;
  label ex;
  var r: String;
  begin @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, ´Mul(´, a, ´,´, b, ´)´);
  @{@&$ENDIF @} checkzero(a, b);
  if ((a[1] = ´-´) ∧ (b[1] ≠ ´-´)) ∨ ((a[1] ≠ ´-´) ∧ (b[1] = ´-´)) then
    begin r ← ´-´ + _Mul(abs(a), abs(b));
    if r = ´-0´ then r ← ´0´;
    end
  else r ← _Mul(abs(a), abs(b));
ex: Mul ← r;
  @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, ´End␣Mul:´, r);
  @{@&$ENDIF @}
  end;
```

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

```
function DivA(a, b : String): String;
  label ex;
  var r: String;
  begin @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, ´DivA(´, a, ´,´, b, ´)´);
  @{@&$ENDIF @} checkzero(a, b);
  if ((a[1] = ´-´) ∧ (b[1] ≠ ´-´)) ∨ ((a[1] ≠ ´-´) ∧ (b[1] = ´-´)) then
    begin r ← ´-´ + _Div(abs(a), abs(b));
    if r = ´-0´ then r ← ´0´;
    end
  else r ← _Div(abs(a), abs(b));
ex: DivA ← r;
  @{@&$IFDEF DEBUGNUM @} WriteLn(infofile, ´End␣DivA:´, r);
  @{@&$ENDIF @}
  end;
```

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

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

```
function IsPrime(a : String): boolean;
  var i: String; r: boolean;
  begin if leq('2', a) then
    begin r ← true; i ← '2';
    while leq(Mul(i, i), a) do
      begin if GCD(a, i) = i then
        begin r ← false; break; end;
      i ← Add(i, '1');
      end;
    end
  else r ← false;
  IsPrime ← r;
  end;
```

159. Divides relation. We can check if "x divides y" by testing if $\gcd(x, y) = |x|$.

```
function Divides(a, b : String): boolean;
  var r: boolean;
  begin r ← GCD(a, b) = abs(a); Divides ← r;
  end;
```

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

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

⟨ Arbitrary-precision rational arithmetic 160 ⟩ ≡

```
procedure RationalReduce(var r : Rational);
  var lGcd: String;
  begin lGcd ← gcd(r.Num, r.Den); r.Num ← diva(r.Num, lGcd); r.Den ← diva(r.Den, lGcd);
  end;
```

This code is used in section 128.

161. Rational addition. We recall

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

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

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

function *RationalSub*(**const** *r1*, *r2*: *Rational*): *Rational*;
  var *lRes*: *Rational*;
  begin *lRes.Num* ← *Sub*(*Mul*(*r1.Num*, *r2.Den*), *Mul*(*r1.Den*, *r2.Num*));
  lRes.Den ← *Mul*(*r1.Den*, *r2.Den*); *RationalReduce*(*lRes*); *RationalSub* ← *lRes*;
  end;

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

function *RationalNeg*(**const** *r1*: *Rational*): *Rational*;
  var *lRes*: *Rational*;
  begin *lRes.Num* ← *Mul*(`´-1´`, *r1.Num*); *lRes.Den* ← *r1.Den*; *RationalNeg* ← *lRes*;
  end;

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

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

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

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

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

function *RationalInv*(**const** *r*: *Rational*): *Rational*;
  var *lRes*: *Rational*;
  begin if *r.Num* ≠ `´0´` **then**
    begin if *le*(*r.Num*, `´0´`) **then** *lRes.Num* ← *Mul*(`´-1´`, *r.Den*)
    else *lRes.Num* ← *r.Den*;
    lRes.Den ← *Abs*(*r.Num*);
    end
  else *lRes* ← *RZero*;
  RationalInv ← *lRes*;
  end;

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

function *RationalDiv*(**const** *r1*, *r2*: *Rational*): *Rational*;
  begin *RationalDiv* ← *RationalMult*(*r1*, *RationalInv*(*r2*));
  end;

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

function *RationalEq*(**const** *r1*, *r2*: *Rational*): *boolean*;
  begin *RationalEq* ← (*r1.Num* = *r2.Num*) ∧ (*r1.Den* = *r2.Den*);
  end;

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

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

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

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

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

⟨ Complex-rational arbitrary-precision arithmetic 169 ⟩ ≡

```
function IsintegerNumber(const z: RComplex): boolean;
  begin IsintegerNumber ← (z.Im.Num = ´0´) ∧ (z.Re.Den = ´1´); end;
function IsNaturalNumber(const z: RComplex): boolean;
  begin IsNaturalNumber ← (z.Im.Num = ´0´) ∧ (z.Re.Den = ´1´) ∧ (geq(z.Re.Num, ´0´)); end;
function IsPrimeNumber(const z: RComplex): boolean;
  begin if IsNaturalNumber(z) ∧ IsPrime(z.Re.Num) then IsPrimeNumber ← true
  else IsPrimeNumber ← false;
  end;
```

This code is used in section 128.

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

```
function AreEqComplex(const z1, z2: RComplex): boolean;
  begin AreEqComplex ← RationalEq(z1.Re, z2.Re) ∧ RationalEq(z1.Im, z2.Im); end;
function IsEqWithInt(const z: RComplex;
                            n: longint): boolean;
  var s: String;
  begin Str(n, s); IsEqWithInt ← (z.Im.Num = ´0´) ∧ (z.Re.Num = s) ∧ (z.Re.Den = ´1´); end;
```

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

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

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

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

173. Adding complex numbers. We compute the sum of $(x_1+\mathrm{i}y_1)$ and $x_2+\mathrm{i}y_2$ to be $(x_1+x_2)+\mathrm{i}(y_1+y_2)$.

function *ComplexAdd*(**const** *z1*, *z2*: *RComplex*): *RComplex*;
  var *lRes*: *RComplex*;
  begin *lRes.Re* ← *RationalAdd*(*z1.Re*, *z2.Re*); *lRes.Im* ← *RationalAdd*(*z1.Im*, *z2.Im*);
  `@{@&$`*IFDEF CH_REPORT* `@}`*CHReport.Out_NumReq3*(*rqRealAdd*, *z1*, *z2*, *lRes*);
  `@{@&$`*ENDIF* `@}`*ComplexAdd* ← *lRes*;
  end;

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

function *ComplexSub*(**const** *z1*, *z2*: *RComplex*): *RComplex*;
  var *lRes*: *RComplex*;
  begin *lRes.Re* ← *RationalSub*(*z1.Re*, *z2.Re*); *lRes.Im* ← *RationalSub*(*z1.Im*, *z2.Im*);
  `@{@&$`*IFDEF CH_REPORT* `@}`*CHReport.Out_NumReq3*(*rqRealDiff*, *z1*, *z2*, *lRes*);
  `@{@&$`*ENDIF* `@}`*ComplexSub* ← *lRes*;
  end;

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

function *ComplexNeg*(**const** *z*: *RComplex*): *RComplex*;
  var *lRes*: *RComplex*;
  begin *lRes.Re* ← *RationalNeg*(*z.Re*); *lRes.Im* ← *RationalNeg*(*z.Im*);
  `@{@&$`*IFDEF CH_REPORT* `@}`*CHReport.Out_NumReq2*(*rqRealNeg*, *z*, *lRes*);
  `@{@&$`*ENDIF* `@}`*ComplexNeg* ← *lRes*;
  end;

176. Multiplying complex numbers. We use the usual formula

$$(x_1+\mathrm{i}y_1)(x_2+\mathrm{i}y_2) = (x_1x_2 - y_1y_2) + \mathrm{i}(x_1y_2 + y_1x_2).$$

function *ComplexMult*(**const** *z1*, *z2*: *RComplex*): *RComplex*;
  var *lRes*: *RComplex*;
  begin if *IsEqWithInt*(*z1*, −1) **then** *ComplexMult* ← *ComplexNeg*(*z2*)
  else if *IsEqWithInt*(*z2*, −1) **then** *ComplexMult* ← *ComplexNeg*(*z1*)
    else begin *lRes.Re* ← *RationalSub*(*RationalMult*(*z1.Re*, *z2.Re*), *RationalMult*(*z1.Im*, *z2.Im*));
      lRes.Im ← *RationalAdd*(*RationalMult*(*z1.Re*, *z2.Im*), *RationalMult*(*z1.Im*, *z2.Re*));
      ComplexMult ← *lRes*;
      `@{@&$`*IFDEF CH_REPORT* `@}`*CHReport.Out_NumReq3*(*rqRealMult*, *z1*, *z2*, *lRes*);
      `@{@&$`*ENDIF* `@}`
      end;
  end;

177. Dividing complex numbers. We recall

$$\frac{x_1 + \mathrm{i}y_1}{x_2 + \mathrm{i}y_2} = \frac{(x_1 + \mathrm{i}y_1)(x_2 - \mathrm{i}y_2)}{x_2^2 + y_2^2}$$

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

```
function ComplexDiv(const z1, z2: RComplex): RComplex;
  var lDenom: Rational; lRes: RComplex;
  begin lRes ← CZero;
  with z2 do lDenom ← RationalAdd(RationalMult(Re, Re), RationalMult(Im, Im));
  if lDenom.Num ≠ ´0´ then
    begin
    lRes.Re ← RationalDiv(RationalAdd(RationalMult(z1.Re, z2.Re), RationalMult(z1.Im, z2.Im)),
                          lDenom);
    lRes.Im ← RationalDiv(RationalSub(RationalMult(z1.Im, z2.Re), RationalMult(z1.Re, z2.Im)),
                          lDenom);
    @{@&$IFDEF CH_REPORT @}CHReport.Out_NumReq3(rqRealDiv, z1, z2, lRes);
    @{@&$ENDIF @}
    end;
  ComplexDiv ← lRes;
  end;
```

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

```
function ComplexInv(const z: RComplex): RComplex;
  begin ComplexInv ← ComplexDiv(COne, z); end;
```

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

```
function ComplexNorm(const z: RComplex): Rational;
  begin ComplexNorm ← RationalAdd(RationalMult(Z.Re, Z.Re), RationalMult(Z.Im, Z.Im)); end;
```

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

```
function CompareInt(X1, X2 : Longint): integer;
  begin if X1 = X2 then CompareInt ← 0
  else if X1 > X2 then CompareInt ← 1
    else CompareInt ← −1;
  end;

function CompareIntStr(X1, X2 : String): integer;
  begin if X1 = X2 then CompareIntStr ← 0
  else if gt(X1, X2) then CompareIntStr ← 1
    else CompareIntStr ← −1;
  end;
```

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

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

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

```
function CompareComplex(const z1, z2: RComplex): integer;
  var lInt: integer;
  begin lInt ← 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 ← CompareIntStr(z1.Im.Den, z2.Im.Den);
  end;
```

File 9

Mizar Objects and Data Structures

182. 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).

⟨ mobjects.pas 182 ⟩ ≡
  ⟨ GNU License 4 ⟩
unit *mobjects*;
  interface
  uses *numbers*;
    ⟨ Public interface for `mobjects.pas` 184 ⟩
  implementation
  mdebug uses *info*;
  end_mdebug
  ⟨ Implementation for `mobjects.pas` 183 ⟩
  end ;

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

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

This code is used in section 182.

184. Constant parameters.

⟨ Public interface for `mobjects.pas` 184 ⟩ ≡
const { Maximum MCollection size }
  MaxSize = 2000000;
  MaxCollectionSize = *MaxSize* **div** *SizeOf*(*Pointer*);
    { Maximum MStringList size }
  MaxListSize = *MaxSize* **div** (*SizeOf*(*Pointer*) ∗ 2);
    { Maximum IntegerList size }
  MaxIntegerListSize = *MaxSize* **div** (*SizeOf*(*integer*));
    { MCollection error codes }
  coIndexError = −1; { Index out of range }
  coOverflow = −2; { Overflow }
  coConsistentError = −3;
  coDuplicate = −5; { Duplicate }
  coSortedListError = −6;
  coIndexExtError = −7;

See also sections 185, 186, 191, 194, 195, 219, 235, 250, 267, 281, 286, 295, 299, 307, 309, 340, 341, 350, 359, 379, 397, 402, 418, 429, 447, and 461.

This code is used in section 182.

185. Type aliases.

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡
type { String pointers }
  PString = ↑*ShortString*;
    { Character set type }
  PCharSet = ↑*TCharSet*;
  TCharSet = **set of** *char*;
    { General arrays }
  PByteArray = ↑*TByteArray*;
  TByteArray = **array** [0 . . 32767] **of** *byte*; { $32767 = 2^{15} - 1$ }
  PWordArray = ↑*TWordArray*;
  TWordArray = **array** [0 . . 16383] **of** *word*; { $16383 = 2^{14} - 1$ }

Section 9.1. BASE OBJECT

186. Object-oriented PASCAL is a bit crufty (like all Object-oriented ALGOL-descended languages).

The base *MObject* "class" has a constructor, destructor, a clone function named *CopyObject*, and a "move" function called *MCopy*.

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡
```
    { MObject base object }
  PObject = ↑MObject;
  ObjectPtr = PObject;
  MObject = object
    constructor Init;
    procedure Free;
    destructor Done; virtual;
    function CopyObject: PObject;
    function MCopy: PObject; virtual;
  end ;
```

187. Note that the *VER70* conditional compilation only plays a role here, in *MObject.Init*.

The constructor will initialize the memory allocated for the *MObject* to be zero.

⟨ *MObject* implementation 187 ⟩ ≡
```
    { MObject }
constructor MObject.Init;
    @{@&$IFDEF VER70 @}
    type Image = record Link: word;
        Data: record
          end;
        end;
    @{@&$ENDIF @}
      begin
      @{@&$IFDEF VER70 @}FillChar(Image(Self).Data, SizeOf(Self) − SizeOf(MObject), 0);
      @{@&$ENDIF @}
      end;
```
This code is used in section 183.

188. Destructor. The *MObject.Free* procdure frees all the memory allocated to the caller.

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

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

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

It then returns a pointer to the newly allocated object.

```
function MObject.CopyObject: PObject;
  var lObject: PObject;
  begin GetMem(lObject, SizeOf(Self)); Move(Self, lObject↑, SizeOf(Self)); CopyObject ← lObject;
  end;
```

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

function *MObject.MCopy*: *PObject*;
  begin *MCopy* $\leftarrow$ *CopyObject*; **end**;

Section 9.2. MIZAR STRING OBJECT

191. Strings in Mizar amount to a wrapper around the underlying string data type.

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡
  { Specyfic objects based on MObjects for collections }
 PStr = ↑*MStrObj*;
 MStrPtr = *PStr*;
 MStrObj = **object** (*MObject*)
  fStr: *String*;
  constructor *Init*(**const** *aStr*: *String*);
 end ;

192. Constructor. The constructor for a string object expects a string, and simply initializes its contents to the given string.

⟨ *MStrObj* implementation 192 ⟩ ≡
  { Specyfic objects based on MObjects for collections }
constructor *MStrObj*.*Init*(**const** *aStr*: *String*);
 begin *fStr* ← *aStr*; **end**;

This code is used in section 183.

Section 9.3. MIZAR LIST

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

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

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

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

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

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

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡

{ MCollection types }
PItemList = ↑*MItemList*;
MItemList = **array** [0 .. *MaxCollectionSize* − 1] **of** *Pointer*;

195. A *MList* object is known as a dynamic array. Java programmers would know thas as an ArrayList.

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡

```
    { MList object }
  PList = ↑MList;
  MListPtr = PList;
  MList = 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 AppendTo(var fAnother : MList); virtual;
    procedure TransferItems(var fAnother : MList); virtual;
    procedure CopyItems(var fOrigin : MList); virtual;
  end ;
```

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

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

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

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

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

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

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

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

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

which means

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

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.

⟨ *MList* implementation 196 ⟩ ≡
  { Simple Collection }
function *GrowLimit*(*aLimit* : *integer*): *integer*;
  begin *GrowLimit* ← 4;
  if *aLimit* > 64 **then** *GrowLimit* ← *aLimit* **div** 4
  else if *aLimit* > 8 **then** *GrowLimit* ← 16;
  end;

This code is used in section 183.

197. Constructor. The constructor creates an empty list.

constructor *MList.Init*(*aLimit* : *integer*);
  begin *MObject.Init*; *Items* ← **nil**; *Count* ← 0; *Limit* ← 0; *SetLimit*(*aLimit*); **end**;

198. Moving a list into the caller.

constructor *MList.MoveList*(**var** *aAnother* : *MList*);
begin *MObject.Init*;
Count ← *aAnother.Count*; *Limit* ← *aAnother.Limit*; *Items* ← *aAnother.Items*; { move }
aAnother.DeleteAll; *aAnother.Limit* ← 0; *aAnother.Items* ← **nil**; { delete *aAnother* }
end;

199. 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.

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

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

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

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

function *MList.MCopy*: *PObject*;
var *lList*: *PObject*; *i*: *integer*;
begin *lList* ← *CopyObject*; *GetMem*(*PList*(*lList*)↑.*Items*, *Self.Limit* ∗ *SizeOf*(*Pointer*));
for *i* ← 0 **to** *Self.Count* − 1 **do** *PList*(*lList*)↑.*Items*↑[*i*] ← *PObject*(*Self.Items*↑[*i*])↑.*MCopy*;
MCopy ← *lList*;
end;

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

function *MList.At*(*Index* : *integer*): *Pointer*;
begin if (*Index* < 0) ∨ (*Index* ≥ *Count*) **then**
begin *ListError*(*coIndexError*, 0); *At* ← **nil**; **end**
else *At* ← *Items*↑[*Index*];
end;

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

function *MList.Last*: *Pointer*;
begin *Last* ← *At*(*Count* − 1); **end**;

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

procedure *MList.Insert*(*aItem* : *Pointer*);
begin if *Limit* = *Count* **then** *SetLimit*(*Limit* + *GrowLimit*(*Limit*));
Items↑[*Count*] ← *aItem*; *inc*(*Count*);
end;

205. If we want to insert a pointer *at a specific index*, then we proceed as follows:
(1) Check if the index is negative. If so, then we should flag an error using *ListError*, and exit.
(2) Check if the index is larger than the logical size of the dynamic array; if so, then we grow the dynamic array using *SetLimit*

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

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

```
procedure MList.InsertList(var aAnother : MList);
  var i: integer;
  begin for i ← 0 to pred(aAnother.Count) do Insert(PObject(aAnother.Items↑[i])↑.MCopy);
  end;
```

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

```
function MList.GetObject(aIndex : integer): Pointer;
  begin if (aIndex < 0) ∨ (aIndex ≥ Count) then
    begin ListError(coIndexError, 0); GetObject ← nil; end
  else GetObject ← Items↑[aIndex];
  end;
```

208. We have a default error code for lists.

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

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

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

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

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

procedure *MList.DeleteAll*;
  begin *Count* $\leftarrow$ 0; **end**;

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

procedure *MList.FreeItem*(*Item* : *Pointer*);
  begin if *Item* $\neq$ **nil then** *Dispose*(*PObject*(*Item*), *Done*);
  end;

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

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

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

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

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

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*↑[i] = **nil then**
      begin for $k \leftarrow i$ **to** *Count* $- 2$ **do** *Items*↑[k] $\leftarrow$ *Items*↑[$k + 1$];
      dec(*Count*);
      end;
  end;

215. Growing a list handles a few edgecases:

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

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

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

procedure $MList.AppendTo($**var** $fAnother : MList)$;
  var k: $integer$;
  begin $SetLimit(Count + fAnother.Count)$;
  for $k \leftarrow 0$ **to** $fAnother.Count - 1$ **do** $Insert(fAnother.Items\uparrow[k])$;
  $fAnother.DeleteAll$; $fAnother.Done$;
  end;

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

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

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

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

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

procedure $MList.CopyItems($**var** $fOrigin : MList)$;
  var i: $integer$;
  begin for $i \leftarrow 0$ **to** $fOrigin.Count - 1$ **do** $Insert(PObject(fOrigin.Items\uparrow[i])\uparrow.CopyObject)$;
  end;

Section 9.4. MIZAR COLLECTION CLASS

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

$$s(n+1) = s(n) + \mathit{GrowLimit}\big(\Delta + s(n)\big)$$

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

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡
  { MCollection object }
PCollection = ↑*MCollection*;
MCollection = **object** (*MList*)
  Delta: *integer*;
  constructor *Init*(*ALimit*, *ADelta* : *integer*);
  destructor *Done*; *virtual*;
  procedure *AtDelete*(*Index* : *integer*);
  procedure *AtFree*(*Index* : *integer*);
  procedure *AtInsert*(*Index* : *integer*; *Item* : *Pointer*); *virtual*;
  procedure *AtPut*(*Index* : *integer*; *Item* : *Pointer*);
  procedure *Delete*(*Item* : *Pointer*);
  procedure *Free*(*Item* : *Pointer*);
  procedure *Insert*(*aItem* : *Pointer*); *virtual*;
  procedure *Pack*; *virtual*;
  constructor *MoveCollection*(**var** *fAnother* : *MCollection*);
  constructor *MoveList*(**var** *aAnother* : *MList*);
  constructor *CopyList*(**var** *aAnother* : *MList*);
  constructor *CopyCollection*(**var** *AAnother* : *MCollection*);
  constructor *Singleton*(*fSing* : *PObject*; *fDelta* : *integer*);
  procedure *Prune*; *virtual*;
end ;

220. Constructor. When constructing a new Collection, we allocate an array of the desired limit (using the *SetLimit* (§215) to handle this allocation).

⟨ *MCollection* implementation 220 ⟩ ≡
  { MCollection }
constructor *MCollection.Init*(*ALimit*, *ADelta* : *integer*);
  begin *MObject.Init*; *Items* ← **nil**; *Count* ← 0; *Limit* ← 0; *Delta* ← *ADelta*; *SetLimit*(*ALimit*);
  end;
destructor *MCollection.Done*;
  begin *FreeAll*; *SetLimit*(0);
  end;

This code is used in section 183.

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

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

procedure *MCollection.AtDelete*(*Index* : *integer*);
  var *i*: *integer*;
  begin if $(Index < 0) \lor (Index \geq Count)$ **then**
    begin *ListError*(*coIndexError*, 0); *exit*; **end**;
  if $Index < pred(Count)$ **then**
    for $i \leftarrow Index$ **to** $Count - 2$ **do** $Items\uparrow[i] \leftarrow Items\uparrow[i+1]$;
  Dec(*Count*);
  end;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

229. Move semantics for creating a new collection.

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

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

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

231. A singleton allocates as little as possible.

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

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

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

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

constructor *MCollection.MoveList*(**var** *aAnother* : *MList*);
 begin *inherited MoveList*(*aAnother*); $Delta \leftarrow 2$;
 end;

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

constructor *MCollection.CopyList*(**var** *aAnother* : *MList*);
 begin *inherited CopyList*(*aAnother*); $Delta \leftarrow 2$; **end**;

Section 9.5. SIMPLE STACKED (EXTENDIBLE) LISTS

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

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

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

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡
    { MExtList object }
  MExtListPtr = ↑*MExtList*;
  MExtList = **object** (*MList*)
    fExtCount: *integer*;
    constructor *Init*(*aLimit* : *integer*);
    destructor *Done*; *virtual*;
    procedure *Insert*(*aItem* : *Pointer*); *virtual*;
    procedure *Mark*(**var** *aIndex* : *integer*); *virtual*;
    procedure *FreeItemsFrom*(*aIndex* : *integer*); *virtual*;
    procedure *DeleteAll*; *virtual*;
    procedure *FreeAll*; *virtual*;
    procedure *Pack*; *virtual*;
    procedure *InsertExt*(*AItem* : *Pointer*); *virtual*;
    procedure *SetLimit*(*ALimit* : *integer*); *virtual*;
    procedure *AddExtObject*; *virtual*;
    procedure *AddExtItems*; *virtual*;
    procedure *DeleteExtItems*;
    procedure *FreeExtItems*;
  end ;

236. Simple Stacked (Extendable) Collection.

⟨ *MExtList* implementation 236 ⟩ ≡
constructor *MExtList.Init*(*ALimit* : *integer*);
  begin *MObject.Init*;
  Items ← **nil**;
  Count ← 0; *Limit* ← 0;
  SetLimit(*ALimit*); *fExtCount* ← 0;
  end;

This code is used in section 183.

237. Destructor for `MExtList`. The destructor for *MExtList* invokes *self.FreeExtItems* and then calls the inherited destructor from the superclass.

destructor *MExtList.Done*;
  begin *FreeExtItems*; *inherited Done*; **end**;

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

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

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

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

```
procedure MExtList.DeleteAll;
  begin if fExtCount ≠ 0 then
    begin ListError(coIndexExtError, 0); exit;
    end;
  inherited DeleteAll;
  end;
```

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

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

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

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

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

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

243. Ensure capacity of extendible list.

(1) When the new limit is less than the logical size N and the extendible size k, we just set the capacity to $N + k$.

(2) Else if the new limit is larger than *MaxCollectionSize*, then just use the maximum collection size as the capacity.

(3) Else if the new limit is different than the existing capacity, then we have to check if the new limit is zero. When it is, just set the capacity to zero and the list of items to **nil**. Otherwise, allocate space for a new array, and move over the contents from the existing array (and then free the existing array). Update the capacity and pointer to the items.

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

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

procedure *MExtList.Mark*(**var** *aIndex* : *integer*);
  begin $aIndex \leftarrow Count$;
  end;

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

procedure *MExtList.FreeItemsFrom*(*aIndex* : *integer*);
  var *I*: *integer*;
  begin if $fExtCount \neq 0$ **then**
    begin *ListError*(*coIndexExtError*, 0); *exit*;
    end;
  for $I \leftarrow Count - 1$ **downto** *aIndex* **do**
    if $Items\uparrow[I] \neq$ **nil then** *Dispose*(*PObject*($Items\uparrow[I]$), *Done*);
  $Count \leftarrow aIndex$;
  end;

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

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

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

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

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

procedure *MExtList.DeleteExtItems*;
  begin $fExtCount \leftarrow 0$;
  end;

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

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

Section 9.6. SORTED LISTS

250. 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 $i = 0, \ldots, Length(Items) - 2$, we have $Items\uparrow[fIndex\uparrow[i]] \le Items\uparrow[fIndex\uparrow[i+1]]$.

Also, we are taking the convention that $fCompare(x, y)$ returns -1 when $x < y$; returns 0 when $x = y$; returns $+1$ when $x > y$.

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡

```
    { MSortedList Object }
  IndexListPtr = ↑MIndexList;
  MIndexList = array [0 .. MaxCollectionSize − 1] of integer;
  CompareProc = function (aItem1, aItem2 : Pointer): integer;
  MSortedList = object (MList)
    fIndex: IndexListPtr;
    fCompare: CompareProc;
    constructor Init(ALimit : integer);
    constructor InitSorted(aLimit : integer; aCompare : CompareProc);
    constructor MoveList(var aAnother : MList);
    constructor CopyList(const aAnother: MList);
    procedure AtInsert(aIndex : integer; aItem : Pointer); virtual;
    procedure Insert(aItem : Pointer); virtual;
    function IndexOf(aItem : Pointer): integer; virtual;
    procedure Sort(aCompare : CompareProc);
    procedure SetLimit(ALimit : integer); virtual;
    function Find(aKey : Pointer; var aIndex : integer): boolean; virtual;
    function Search(aKey : Pointer; var aIndex : integer): boolean; virtual;
    procedure Pack; virtual;
    procedure FreeItemsFrom(aIndex : integer); virtual;
  end ;
```

251. Constructors. There are four constructors:

(1) *Init* simply creates an empty list with a given capacity.
(2) *InitSorted* is like *Init*, but expects an ordering operator.
(3) *MoveList* moves all the items from another list into the caller, sorting as needed. This will also empty the other list.
(4) *CopyList* is like *MoveList* but leaves the other list untouched.

⟨ *MSortedList* implementation 251 ⟩ ≡

```
    { MSortedList object }
constructor MSortedList.Init(aLimit : integer);
  begin MObject.Init; Items ← nil; Count ← 0; Limit ← 0; fIndex ← nil; fCompare ← nil;
  SetLimit(ALimit);
  end;
constructor MSortedList.InitSorted(aLimit : integer; aCompare : CompareProc);
  begin MObject.Init; Items ← nil; Count ← 0; Limit ← 0; fIndex ← nil; fCompare ← aCompare;
  SetLimit(ALimit);
  end;
```

See also section 263.

This code is used in section 183.

252. Move constructor. When we move items from an *MList* into the caller, we also sort as we insert.

```
constructor MSortedList.MoveList(var aAnother : MList);
  var I: integer;
  begin Items ← aAnother.Items; Count ← aAnother.Count; Limit ← aAnother.Limit;
  GetMem(fIndex, Limit ∗ SizeOf(integer)); fCompare ← nil;
  for I ← 0 to pred(aAnother.Count) do fIndex↑[I] ← I;
      { Empty out the other list }
  aAnother.DeleteAll; aAnother.Limit ← 0; aAnother.Items ← nil;
  end;
```

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

```
constructor MSortedList.CopyList(const aAnother: MList);
  var i: integer;
  begin MObject.Init; Items ← nil; Count ← 0; Limit ← 0; fIndex ← nil; fCompare ← nil;
  SetLimit(aAnother.Limit); Count ← aAnother.Count;
  for i ← 0 to Count − 1 do
    begin Items↑[i] ← PObject(aAnother.Items↑[i])↑.MCopy; fIndex↑[I] ← I;
    end;
  end;
```

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

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

255. Inserting an item. Inserting an item into a sorted list boils down to two cases:

(1) If there is an ordering operator, we check if the item is in the underlying array using *Find* (§260), which will mutate the *lIndex* to be where it should be located. When the item is missing, simply insert it at *lIndex*. When the item is present, then we do nothing.

(2) If there is no ordering operator, then check if the item already is present in the sorted list. If so, then don't do annything. Otherwise, insert the item at the start of the list.

```
procedure MSortedList.Insert(aItem : Pointer);
  var lIndex: integer;
  begin if @fCompare = nil then
    begin if Limit = Count then SetLimit(Limit + GrowLimit(Limit));
    Items↑[Count] ← aItem; fIndex↑[Count] ← Count; inc(Count); exit;
    end;
  if ¬Find(aItem, lIndex) then AtInsert(lIndex, aItem);
  end;
```

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

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

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

Algorithm S (*Quicksort*). This uses Hoare partition. We assume that $L \le R$, 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 $P \leftarrow aList\uparrow[aIndex\uparrow[(L + R)\ \mathbf{shr}\ 1]]$. Observe $I \le P_{idx} \le J$ at this point.

S1. [Move I right] While $aList[I] < P$, we increment $I \leftarrow I + 1$. This is guaranteed to terminate since $I \le P_{idx}$, so eventually we will get to $aList[i] = P$.

S2. [Move J left] While $P < aList[J]$, we decrement $J \leftarrow J - 1$. This is guaranteed to terminate since $P_{idx} \le J$, so eventually we will get to $aList[J] = P$.

S3. [Keep going?] If $I > J$, then we're done "partitioning" (so everything to the left of the pivot is not greater than the pivot value, and everything to the right of the pivot is not lesser than the pivot value), and we go to step S5; otherwise go to the next step.

S4. [Swap entries I and J] We swap the entries located at I and J, then set $I \leftarrow I + 1$, and $J \leftarrow J - 1$. If $I \le J$, 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 $L \mathinner{\ldotp\ldotp} J - 1$).

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 $swap\_indices(\#) \equiv T \leftarrow aIndex\uparrow[\#];\ aIndex\uparrow[\#] \leftarrow steal\_from$

```
procedure ListQuickSort(aList : PItemList; aIndex : IndexListPtr; L, R : integer;
        aCompare : CompareProc);
  var I, J, T: integer; P: Pointer;
  begin repeat I ← L; J ← R; P ← aList↑[aIndex↑[(L + R) shr 1]];
    repeat
      { I ≤ (L + R) shr 1 ≤ J }
      while aCompare(aList↑[aIndex↑[I]], P) < 0 do inc(I);
      { P ≤ aList↑[aIndex↑[I]] }
      while aCompare(aList↑[aIndex↑[J]], P) > 0 do Dec(J);
      { aList↑[aIndex↑[J]] ≤ P }
      { I ≤ (L + R) shr 1 ≤ J }
      if I ≤ J then
        begin
        { aList↑[aIndex↑[J]] < P < aList↑[aIndex↑[I]] }
        swap_indices(I)(J);
        { aList↑[aIndex↑[I]] < P < aList↑[aIndex↑[J]] }
        { I < J implies inc(I) ≤ dec(J) }
        { I = J implies inc(I) > dec(J) }
        inc(I); Dec(J);
        end;
    until I > J;
    { J ≤ (L + R) shr 2 ≤ I and J < I }
    if L < J then ListQuickSort(aList, aIndex, L, J, aCompare);  { quicksort left half }
    L ← I;  { recursively quicksort the right half of the array }
  until I ≥ R;
  end;
```

258. 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 $a \leq b \wedge b \leq a \implies a = b$ (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 $L \mathrel{..} J-1$ are partitioned (i.e., less than the pivot value) as is the entries in $I \mathrel{..} R$. In particular, the maximal element in $L \mathrel{..} J-1$ is located at $J-1$ while the minimal element in $I \mathrel{..} R$ is located at I.

(4) Robert Sedgewick's *Quicksort* (1980) is literally *the* book on the subject. An abbreviated reference may be found in Sedgewick's "The Analysis of Quicksort Programs" (*Acta Inform.* **7** (1977) 327–355, eprint)

(5) IMPROVEMENT: This can be improved when recursively sorting the left half of the arrays by first checking if $J - L \leq 9$ then use insertion sort otherwise recursively quicksort the left half. (Similarly, instead of iterating the outermost while-loop, we should test if $R - I \leq 9$ then invoking insertion on the subarray indexed by $I \mathrel{..} R$.)

(6) IMPROVEMENT: The pivot index P_{ind} is selected as $P_{ind} \leftarrow (L+R)/2$, which can lead to overflow. A safer way to compute this would be $P_{ind} \leftarrow L + ((R-L)/2)$.

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

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

We can use the approximation for Harmonic numbers

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

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

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

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

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

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

procedure *MSortedList.Sort*(*aCompare* : *CompareProc*);
  var *I*: *integer*;
  begin *fCompare* ← *aCompare*;
  for *I* ← 0 **to** *Count* − 1 **do** *fIndex*↑[*I*] ← *I*;
  if (*Count* > 0) **then** *ListQuickSort*(*Items*, *fIndex*, 0, *Count* − 1, *aCompare*);
  end;

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

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

function *MSortedList.Find*(*aKey* : *Pointer*; **var** *aIndex* : *integer*): *boolean*;
  var L, H, I, C: *integer*;
  begin *Find* ← *False*;
  if @*fCompare* = **nil** **then** ⟨Find needle in *MSortedList* by brute force 262⟩;
  ⟨Find needle in *MSortedList* by binary search 261⟩
  end;

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

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

On the other hand, if $C \geq 0$, 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.

⟨Find needle in *MSortedList* by binary search 261⟩ ≡
  $L \leftarrow 0$; $H \leftarrow Count - 1$;
  while $L \leq H$ **do**
    begin $I \leftarrow (L + H)$ **shr** 1; $C \leftarrow$ *fCompare*(*Items*↑[*fIndex*↑[I]], *aKey*);
    if $C < 0$ **then** $L \leftarrow I + 1$
    else begin $H \leftarrow I - 1$;
      if $C = 0$ **then**
        begin *Find* ← *True*; $L \leftarrow I$; **end**;
      end;
    end;
  aIndex ← L;

This code is used in section 260.

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

⟨Find needle in *MSortedList* by brute force 262⟩ ≡
  begin *aIndex* ← *Count*;
  for $I \leftarrow 0$ **to** $Count - 1$ **do**
    if *aKey* = *Items*↑[I] **then**
      begin *Find* ← *True*; *aIndex* ← I; *break* **end**;
  exit;
  end

This code is used in section 260.

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

⟨ *MSortedList* implementation 251 ⟩ +≡
function *MSortedList.Search*(*aKey* : *Pointer*; **var** *aIndex* : *integer*): *boolean*;
  var *I*: *integer*;
  begin *aIndex* ← *Count*; *Search* ← *false*;
  if *Find*(*aKey*, *I*) **then**
    begin *Search* ← *true*; *aIndex* ← *fIndex*↑[*I*];
    end;
  end;

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

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

function *MSortedList.IndexOf*(*aItem* : *Pointer*): *integer*;
  var *I*: *integer*;
  begin if @*fCompare* = **nil then**
    begin *ListError*(*coSortedListError*, 0); *exit*;
    end;
  IndexOf ← −1;
  if *Find*(*aItem*, *I*) **then**
    begin { if $I <$ *fCount* then }
    IndexOf ← *fIndex*↑[*I*];
    end;
  end;

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

procedure *MSortedList.Pack*;
  var *lCount*: *integer*;
  begin *lCount* ← *Count*; *inherited Pack*;
  if (@*fCompare* ≠ **nil**) ∧ (*lCount* > *Count*) **then** *Sort*(*fCompare*);
  end;

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

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

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

Section 9.7. SORTED EXTENDIBLE LISTS

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

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

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

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

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

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

⟨ *MSortedExtList* implementation 268 ⟩ ≡
    { MSortedExtList always with possible duplicate keys, always sorted }
constructor *MSortedExtList.Init*(*ALimit* : *integer*);
  begin *ListError*(*coIndexExtError*, 0); **end**;

constructor *MSortedExtList.InitSorted*(*aLimit* : *integer*; *aCompare* : *CompareProc*);
  begin *inherited Init*(*aLimit*); *fCompare* ← *aCompare*;
  end;

This code is used in section 183.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

function *MSortedExtList.AtIndex*(*aIndex* : *integer*): *Pointer*;
  begin if $(aIndex < 0) \lor (aIndex \geq Count)$ **then** *ListError*(*coIndexExtError*, 0);
  AtIndex ← *Items*↑[*fIndex*↑[*aIndex*]];
  end;

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

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

procedure *MSortedExtList.Insert*(*aItem* : *Pointer*);
  begin if $fExtCount \neq 0$ **then** *ListError*(*coIndexExtError*, 0);
  InsertExt(*aItem*); *AddExtObject*;
  end;

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

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

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

procedure *MSortedExtList.InsertExt*(*AItem* : *Pointer*);
  begin if $Limit = Count + fExtCount$ **then** *SetLimit*(*Limit* + *GrowLimit*(*Limit*));
  Items↑[*Count* + *fExtCount*] ← *AItem*; *inc*(*fExtCount*);
  end;

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

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

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

278. Freeing items starting at an index. We have two exceptional situations:
(1) The *fExtCount* must be zero, and if it is nonzero, then an error is raised; and
(2) If the index given is equal to the logical size of the sorted extendible list, then we terminate early (since there is nothing to do).

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

279. Digest an extendible object. When there are extendible objects left to digest among the values (i.e., when $fExtCount > 0$), When $fExtCount \leq 0$, 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.

procedure *MSortedExtList.AddExtObject*;
  var *lIndex*: *integer*;
  begin if $fExtCount \leq 0$ **then** *ListError*(*coIndexExtError*, 0);
  FindRight(*Items*↑[*Count*], *lIndex*);
  if $lIndex \neq Count$ **then** { shift *fIndex* to right by 1 }
    Move($fIndex\uparrow[lIndex], fIndex\uparrow[lIndex + 1], (Count - lIndex) * SizeOf(integer)$);
  $fIndex\uparrow[lIndex] \leftarrow Count$; { extendible item's index }
  inc(*Count*); *Dec*(*fExtCount*);
  end;

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

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

Section 9.8. SORTED LIST OF STRINGS

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

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡
  MSortedStrList = **object** (*MSortedList*)
    constructor *Init*(*ALimit* : *integer*);
    function *IndexOfStr*(**const** *aStr*: *String*): *integer*; *virtual*;
    function *ObjectOf*(**const** *aStr*: *String*): *PObject*; *virtual*;
  end ;

282. Pointer comparison. For strings, it is faster to use pointer comparison than lexicographic ordering. Although pointer comparison is a total linear order, it may not produce intuitive comparisons.

⟨ *MSortedStrList* implementation 282 ⟩ ≡
    { MSortedStrList }
function *CompareStringPtr*(*aKey1*, *aKey2* : *Pointer*): *integer*;
  begin if *PStr*(*aKey1*)↑.*fStr* < *PStr*(*aKey2*)↑.*fStr* **then** *CompareStringPtr* ← −1
  else if *PStr*(*aKey1*)↑.*fStr* = *PStr*(*aKey2*)↑.*fStr* **then** *CompareStringPtr* ← 0
    else *CompareStringPtr* ← 1;
  end;

This code is used in section 183.

283. Constructor. We just defer to the *InitSorted* constructor for sorted lists (§251).

As an invariant, the *fCompare* ordering operator is *always* assumed to be set to the *CompareStringPtr*. There is no other way to construct a sorted string list besides this constructor, which enforces this invariant.

constructor *MSortedStrList*.*Init*(*ALimit* : *integer*);
  begin *InitSorted*(*ALimit*, *CompareStringPtr*);
  end;

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

function *MSortedStrList*.*IndexOfStr*(**const** *aStr*: *string*): *integer*;
  var *I*: *integer*; *lStringObj*: *MStrObj*;
  begin *IndexOfStr* ← −1;
  if @*fCompare* = **nil then**   { Invariant violation }
    begin *ListError*(*coSortedListError*, 0); *exit*;
    end;
  lStringObj.*Init*(*aStr*);
  if *Find*(@*lStringObj*, *I*) **then**
    begin if *I* < *Count* **then** *IndexOfStr* ← *fIndex*↑[*I*];
    end;
  end;

285. We also can return the pointer to the object, if it is present in the sorted string list.

function *MSortedStrList*.*ObjectOf*(**const** *aStr*: *string*): *PObject*;
  var *I*: *integer*;
  begin *ObjectOf* ← **nil**; *I* ← *IndexOfStr*(*aStr*);
  if *I* ≥ 0 **then** *ObjectOf* ← *Items*↑[*I*];
  end;

Section 9.9. SORTED COLLECTIONS

286.

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡
{ 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 ;

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

⟨ *MSortedCollection* implementation 287 ⟩ ≡
{ MSortedCollection }
constructor *MSortedCollection*.*Init*(*aLimit*, *aDelta* : *integer*);
begin *inherited Init*(*ALimit*, *ADelta*); *Duplicates* ← *False*; *fCompare* ← **nil**;
end;

constructor *MSortedCollection*.*InitSorted*(*aLimit*, *aDelta* : *integer*; *aCompare* : *CompareProc*);
begin *inherited Init*(*ALimit*, *ADelta*); *Duplicates* ← *False*; *fCompare* ← *aCompare*;
end;

This code is used in section 183.

288. Comparing entries. This will invoke *Abstract1* (§183) when there is no ordering operator, which itself raises an error 211.

Otherwise, this just invokes *fCompare* on the two entries.

function *MSortedCollection*.*Compare*(*Key1*, *Key2* : *Pointer*): *integer*;
begin if @*fCompare* = **nil then** *Abstract1*;
Compare ← *fCompare*(*Key1*, *Key2*);
end;

289. Find the right-most index for an item in the collection. Searching (§293) for the *KeyOf* (§292). This function corrects for the possible bug in *Search* (§293) which will return the index *just to the left* of an item.

function *MSortedCollection*.*IndexOf*(*aItem* : *Pointer*): *integer*;
var *I*: *integer*;
begin *IndexOf* ← −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* ← *I*;
end;
end;

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

procedure *MSortedCollection.Insert*(*aItem* : *Pointer*);
  var I: *integer*;
  begin if $\neg$*Search*(*KeyOf*(*aItem*), I) $\vee$ *Duplicates* **then** *AtInsert*(I, *aItem*);
  end;

291. Insert an item if it's not in the collection (or if there are duplicates allowed in the collection). Otherwise, delete the item and do not mutate the caller.

procedure *MSortedCollection.InsertD*(*Item* : *Pointer*);
  var I: *integer*;
  begin if $\neg$*Search*(*KeyOf*(*Item*), I) $\vee$ *Duplicates* **then** *AtInsert*(I, *Item*)
  else *Dispose*(*PObject*(*Item*), *Done*);
  end;

292. We treat the item itself as the key, so return the item. That is to say, this is the identity function. It does not mutate the caller.

function *MSortedCollection.KeyOf*(*Item* : *Pointer*): *Pointer*;
  begin *KeyOf* $\leftarrow$ *Item*;
  end;

293. Binary search. This is binary search through a sorted collection.
If there are duplicates, this will return the left-most index.

function *MSortedCollection.Search*(*Key* : *Pointer*; **var** *Index* : *integer*): *boolean*;
  var L, H, I, C: *integer*;
  begin *Search* $\leftarrow$ *False*; $L \leftarrow 0$; $H \leftarrow$ *Count* $- 1$;
  while $L \le H$ **do**
    begin $I \leftarrow (L + H)$ **shr** 1; $C \leftarrow$ *Compare*(*KeyOf*(*Items*↑[I]), *Key*);
    if $C < 0$ **then** $L \leftarrow I + 1$
    else begin $H \leftarrow I - 1$;
      if $C = 0$ **then**
        begin *Search* $\leftarrow$ *True*;
        if $\neg$*Duplicates* **then** $L \leftarrow I$;
        end;
      end;
    end;
  Index $\leftarrow L$;
  end;

294. 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;

Section 9.10. STRING COLLECTION

295.

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡
{ MStringCollection object }
PStringCollection = ↑*MStringCollection*;
MStringCollection = **object** (*MSortedCollection*)
function *Compare*(*Key1*, *Key2* : *Pointer*): *integer*; *virtual*;
procedure *FreeItem*(*Item* : *Pointer*); *virtual*;
end ;
{ UnsortedStringCollection }
PUsortedStringCollection = ↑*StringColl*;
StringColl = **object** (*MCollection*)
procedure *FreeItem*(*Item* : *pointer*); *virtual*;
end ;

296. String ordering operator. We have the usual lexicograph ordering as an operator ordering.

⟨ String collection implementation 296 ⟩ ≡
{ MStringCollection }
function *CompareStr*(*aStr1*, *aStr2* : *string*): *integer*;
begin if *aStr1* < *aStr2* **then** *CompareStr* ← −1
else if *aStr1* = *aStr2* **then** *CompareStr* ← 0
else *CompareStr* ← 1;
end;

This code is used in section 183.

297. We then have a convenience function to handle pointer dereferencing.

function *MStringCollection*.*Compare*(*Key1*, *Key2* : *Pointer*): *integer*;
begin *Compare* ← *CompareStr*(*PString*(*Key1*)↑, *PString*(*Key2*)↑);
end;

298. 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;

Section 9.11. INT COLLECTIONS

299. The *TIntItem* is needed for the unifier and equalizer.

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡
{ MIntCollection object }
IntPair = **record** *X*, *Y*: *integer*;
end;
IntPairItemPtr = ↑*IntPairItem*;
IntPairItem = **object** (*MObject*)
fKey: *IntPair*;
constructor *Init*(*X*, *Y* : *integer*);
end ;
IntPtr = ↑*integer*;
PIntItem = ↑*TIntItem*;
TIntItem = **object** (*MObject*)
IntKey: *integer*;
constructor *Init*(*fInt* : *integer*);
end ;
PIntKeyCollection = ↑*TIntKeyCollection*;
TIntKeyCollection = **object** (*MSortedCollection*)
function *KeyOf*(*Item* : *pointer*): *pointer*; *virtual*;
function *Compare*(*Key1*, *Key2* : *pointer*): *integer*; *virtual*;
end ;
IntPairKeyCollectionPtr = ↑*IntPairKeyCollection*;
IntPairKeyCollection = **object** (*MSortedCollection*)
function *Compare*(*Key1*, *Key2* : *pointer*): *integer*; *virtual*;
function *ObjectOf*(*X*, *Y* : *integer*): *IntPairItemPtr*; *virtual*;
function *FirstThat*(*X* : *integer*): *IntPairItemPtr*; *virtual*;
end ;

300. TIntItem constructor. This just copies the given integer over to the newly allocated *TIntItem* object.

⟨ *MIntCollection* implementation 300 ⟩ ≡
{ MIntCollection }
constructor *TIntItem.Init*(*fInt* : *integer*);
begin *IntKey* ← *fInt*;
end;

This code is used in section 183.

301. We use *TIntItem*s as keys in a *TIntKeyCollection*.

function *TIntKeyCollection.KeyOf*(*Item* : *pointer*): *pointer*;
begin *KeyOf* ← *addr*(*PIntItem*(*Item*)↑.*IntKey*);
end;

302. Comparing items just looks at the integers referred by the pointers.

function *TIntKeyCollection.Compare*(*Key1*, *Key2* : *pointer*): *integer*;
  begin *Compare* ← 1;
  if *IntPtr*(*Key1*)↑ < *IntPtr*(*Key2*)↑ **then**
    begin *Compare* ← −1; *exit*
    end;
  if *IntPtr*(*Key1*)↑ = *IntPtr*(*Key2*)↑ **then** *Compare* ← 0;
  end;

303. Constructor for pairs of integers.

constructor *IntPairItem.Init*(X, Y : *integer*);
  begin *fKey.X* ← X; *fKey.Y* ← Y;
  end;

304. Comparing two keys in a collection indexed by *IntPair*s is done "in the obvious way".

function *IntPairKeyCollection.Compare*(*Key1*, *Key2* : *pointer*): *integer*;
  begin *Compare* ← *CompareIntPairs*(*IntPairItemPtr*(*Key1*)↑.*fKey.X*, *IntPairItemPtr*(*Key1*)↑.*fKey.Y*,
      IntPairItemPtr(*Key2*)↑.*fKey.X*, *IntPairItemPtr*(*Key2*)↑.*fKey.Y*);
  end;

305. We can lookup the value associated to the key (X, Y) leveraging the *MSortedCollection.Search* function.

function *IntPairKeyCollection.ObjectOf*(X, Y : *integer*): *IntPairItemPtr*;
  var *lPairItem*: *IntPairItem*; I: *integer*;
  begin *ObjectOf* ← **nil**; *lPairItem.Init*(X, Y);
  if *Search*(*addr*(*lpairItem*), I) **then** *ObjectOf* ← *Items*↑[I];
  end;

306. This is used in `justhan.pas` and `mizprep.pas`.

function *IntPairKeyCollection.FirstThat*(X : *integer*): *IntPairItemPtr*;
  var I: *integer*;
  begin *FirstThat* ← **nil**;
  for i ← 0 **to** *Count* − 1 **do**
    if *IntPairItemPtr*(*Items*↑[I])↑.*fKey.X* = X **then**
      begin *FirstThat* ← *Items*↑[I]; *exit*
      end;
  end;

Section 9.12. STACKED LIST OF OBJECTS

307. "Stacked" lists are really linked lists.

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡
{ Stacked Object (List of objects) }
StackedPtr = ↑*StackedObj*;
StackedObj = **object** (*MObject*)
Previous: *StackedPtr*;
constructor *Init*;
destructor *Done*; *virtual*;
end ;

308. The constructors and destructors are not implemented, so if you try to use them, just raise an error.

⟨ Stacked object implementation 308 ⟩ ≡
{ Stacked Object (List of objects) }
constructor *StackedObj.Init*;
begin *Abstract1*;
end;
destructor *StackedObj.Done*;
begin *Abstract1*;
end;

This code is used in section 183.

Section 9.13. STRING LIST

309.

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡

{ MStringList object }

MDuplicates = (*dupIgnore*, *dupAccept*, *dupError*);

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

const *aStr*: *String*;
aObject: *PObject*);
end ;

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

⟨ String list implementation 310 ⟩ ≡
{ ———— MStringList ———— }
constructor *MStringList.Init*(*aCapacity* : *integer*);
begin *MObject.Init*; *fList* ← **nil**; *fCount* ← 0; *fCapacity* ← 0; *fSorted* ← *false*;
fDuplicate ← *dupError*; *SetCapacity*(*aCapacity*);
end;
constructor *MStringList.MoveStringList*(**var** *aAnother* : *MStringList*);
begin *MObject.Init*; *fCount* ← *aAnother.fCount*; *fCapacity* ← *aAnother.fCapacity*;
fSorted ← *aAnother.fSorted*; *fList* ← *aAnother.fList*; *fDuplicate* ← *aAnother.fDuplicate*;
{ Empty out the other list }
aAnother.fCount ← 0; *aAnother.fCapacity* ← 0; *aAnother.fList* ← **nil**;
end;

See also section 315.

This code is used in section 183.

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

destructor *MStringList.Done*;
var *I*: *integer*;
begin *inherited Done*;
for *I* ← 0 **to** *fCount* − 1 **do**
with *fList*↑[*I*] **do** { free *fList*↑[*I*] }
begin *DisposeStr*(*fString*);
if *fObject* ≠ **nil then** *Dispose*(*fObject*, *Done*);
end;
fCount ← 0; *SetCapacity*(0);
end;

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

function *MStringList.AddString*(**const** *aStr*: *string*): *integer*;
var *lResult*: *integer*;
begin ⟨ Set *lResult* to the index of the newly inserted string 313 ⟩;
InsertItem(*lResult*, *aStr*); *AddString* ← *lResult*;
end;

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

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

⟨ Set *lResult* to the index of the newly inserted string 313 ⟩ ≡
  if ¬*fSorted* **then** *lResult* ← *fCount*
  else if *Find*(*aStr*, *lResult*) **then**
    begin *AddString* ← *lResult*; ⟨ De-duplicate a string list 314 ⟩;
    end

This code is used in section 312.

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

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

All other situations "fall through".

⟨ De-duplicate a string list 314 ⟩ ≡
  case *fDuplicate* **of**
  dupIgnore: *Exit*;
  dupError: *StringListError*(*coDuplicate*, 0);
  endcases

This code is used in section 313.

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

⟨ String list implementation 310 ⟩ +≡
function *MStringList.AddObject*(**const** *aStr*: *string*; *aObject*: *PObject*): *integer*;
  var *lResult*: *integer*;
  begin *lResult* ← *AddString*(*aStr*); { Insert key }
  PutObject(*lResult*, *aObject*); { Insert value }
  AddObject ← *lResult*; { Return index }
  end;

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

procedure *MStringList.AddStrings*(**var** *aStrings* : *MStringList*);
  var *I*, *r*: *integer*;
  begin for *I* ← 0 **to** *aStrings.fCount* − 1 **do**
    r ← *AddObject*(*aStrings.fList*↑[*I*].*fString*↑, *aStrings.fList*↑[*I*].*fObject*);
  end;

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

procedure *MStringList.Clear*;
  var *I*: *integer*;
  begin if *fCount* ≠ 0 **then**
    begin for *I* ← 0 **to** *fCount* − 1 **do** *DisposeStr*(*fList*↑[*I*].*fString*);
    fCount ← 0; *SetCapacity*(0);
    end;
  end;

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

procedure *MStringList.Delete*(*aIndex* : *integer*);
  begin if $(aIndex < 0) \lor (aIndex \geq fCount)$ **then** *StringListError*(*coIndexError*, *aIndex*);
  DisposeStr(*fList*↑[*aIndex*].*fString*); *Dec*(*fCount*);
  if $aIndex < fCount$ **then**
    Move(*fList*↑[*aIndex* + 1], *fList*↑[*aIndex*], (*fCount* − *aIndex*) ∗ *SizeOf*(*MStringItem*));
  end;

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

procedure *MStringList.Exchange*(*Index1*, *Index2* : *integer*);
  begin if $(Index1 < 0) \lor (Index1 \geq fCount)$ **then** *StringListError*(*coIndexError*, *Index1*);
  if $(Index2 < 0) \lor (Index2 \geq fCount)$ **then** *StringListError*(*coIndexError*, *Index2*);
  ExchangeItems(*Index1*, *Index2*);
  end;

procedure *MStringList.ExchangeItems*(*Index1*, *Index2* : *integer*);
  var *Temp*: *MStringItem*;
  begin *Temp* ← *fList*↑[*Index1*]; *fList*↑[*Index1*] ← *fList*↑[*Index2*]; *fList*↑[*Index2*] ← *Temp*;
  end;

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

function *MStringList.Find* (**const** *aStr*: *string*; **var** *aIndex*: *integer*) : *boolean*;
  var L, H, I, C: *integer*; *lResult*: *boolean*;
  begin *lResult* ← *False*; $L \leftarrow 0$; $H \leftarrow fCount - 1$;
  while $L \leq H$ **do**
    begin $I \leftarrow (L + H)$ **shr** 1; $C \leftarrow$ *CompareStr*(*fList*↑[*I*].*fString*↑, *aStr*);
    if $C < 0$ **then** $L \leftarrow I + 1$
    else begin $H \leftarrow I - 1$;
      if $C = 0$ **then**
        begin *lResult* ← *True*;
        if *fDuplicate* ≠ *dupAccept* **then** $L \leftarrow I$;
        end;
      end;
    end;
  aIndex ← *L*; *Find* ← *lResult*;
  end;

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

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

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

function *MStringList.GetString*(*aIndex* : *integer*): *string*;
  begin if $(aIndex < 0) \lor (aIndex \geq fCount)$ **then** *StringListError*(*coIndexError*, *aIndex*);
  GetString ← ´´;
  if *fList*↑[*aIndex*].*fString* ≠ **nil then** *GetString* ← *fList*↑[*aIndex*].*fString*↑;
  end;

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

```
function MStringList.GetObject(aIndex : integer): PObject;
  begin if (aIndex < 0) ∨ (aIndex ≥ fCount) then StringListError(coIndexError, aIndex);
  GetObject ← fList↑[aIndex].fObject;
  end;
```

324. Ensure capacity for string lists. The growth rate for string lists differs from the earlier discussion on the growth rate for dynamic arrays. Well, actually, recalling our discussion (§196), 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 ← fCapacity div 4
  else if fCapacity > 8 then Delta ← 16
    else Delta ← 4;
  SetCapacity(fCapacity + Delta);
  end;
```

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

```
function MStringList.IndexOf(const aStr: string): integer;
  var lResult: integer;
  begin if ¬fSorted then
    begin for lResult ← 0 to fCount − 1 do
      if CompareStr(fList↑[lResult].fString↑, aStr) = 0 then
        begin IndexOf ← lResult; Exit; end;
    lResult ← −1;
    end
  else if ¬Find(aStr, lResult) then lResult ← −1;
  { Assert: lResult = −1 if aStr is not in the caller }
  IndexOf ← lResult;
  end;
```

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

```
function MStringList.ObjectOf(const aStr: string): PObject;
  var I: integer;
  begin ObjectOf ← nil; I ← IndexOf(aStr);
  if I ≥ 0 then ObjectOf ← fList↑[I].fObject;
  end;
```

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

```
procedure MStringList.Insert(aIndex : integer;
    const aStr: string);
  begin if fSorted then StringListError(coSortedListError, 0);
  if (aIndex < 0) ∨ (aIndex > fCount) then StringListError(coIndexError, aIndex);
  InsertItem(aIndex, aStr);
  end;
```

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

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

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

function *MStringList.IndexOfObject*(*aObject* : *PObject*): *integer*;
 var *lResult*: *integer*;
 begin for $lResult \leftarrow 0$ **to** $fCount - 1$ **do**
  if $GetObject(lResult) = aObject$ **then**
   begin $IndexOfObject \leftarrow lResult$; *Exit*; **end**;
 $IndexOfObject \leftarrow -1$;
 end;

330. Inserting an object associated with a string.

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

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

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

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

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

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

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

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

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

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

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

procedure *MStringList.QuickSort*(*L*, *R* : *integer*);
 var *I*, *J*: *integer*; *P*: *string*;
 begin repeat $I \leftarrow L$; $J \leftarrow R$; $P \leftarrow fList\uparrow[(L + R)$ **shr** $1].fString\uparrow$;
  repeat while $CompareStr(fList\uparrow[I].fString\uparrow, P) < 0$ **do** *inc*(*I*);
   while $CompareStr(fList\uparrow[J].fString\uparrow, P) > 0$ **do** *Dec*(*J*);
   if $I \leq J$ **then**
    begin *ExchangeItems*(*I*, *J*); *inc*(*I*); *Dec*(*J*); **end**;
  until $I > J$;
  if $L < J$ **then** *QuickSort*(*L*, *J*);
  $L \leftarrow I$;
 until $I \geq R$;
 end;

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

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

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

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

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

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

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

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

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

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

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

Section 9.14. TUPLES OF INTEGERS

340.

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡
{ Partial integers Functions }
IntTriplet = **record** *X1*, *X2*, *Y*: *integer*;
end;
const *MaxIntPairSize* = *MaxSize* **div** *SizeOf* (*IntPair*);
MaxIntTripletSize = *MaxSize* **div** *SizeOf* (*IntTriplet*);

341. Now, this is the remainder of the interface

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡
type *IntPairListPtr* = ↑*IntPairList*; *IntPairList* = **array** [0 . . *MaxIntPairSize* − 1] **of** *IntPair*;
IntPairSeqPtr = ↑*BinIntFunc*; *IntPairSeq* = **object** (*MObject*)
Items: *IntPairListPtr*;
Count: *integer*;
Limit: *integer*;
constructor *Init*(*aLimit* : *integer*);
procedure *NatSetError*(*Code*, *Info* : *integer*); *virtual*;
destructor *Done*; *virtual*;
procedure *SetLimit*(*aLimit* : *integer*); *virtual*;
procedure *Insert*(**const** *aItem*: *IntPair*); *virtual*;
procedure *AtDelete*(*aIndex* : *integer*);
procedure *DeleteAll*;
procedure *AssignPair*(*X*, *Y* : *integer*); *virtual*;
end ;

342. First, we have a helper function for flagging errors.

⟨ Tuples of integers 342 ⟩ ≡
{ Pairs of an integers }
procedure *IntPairSeq.NatSetError*(*Code*, *Info* : *integer*);
begin *RunError*(212 − *Code*); **end**;

343. Constructor.

constructor *IntPairSeq.Init*(*aLimit* : *integer*);
begin *MObject.Init*; *Items* ← **nil**; *Count* ← 0; *Limit* ← 0; *SetLimit*(*aLimit*);
end;

344. Destructor.

destructor *IntPairSeq.Done*;
begin *Count* ← 0; *SetLimit*(0);
end;

345. Insert an element.

procedure *IntPairSeq.Insert*(**const** *aItem*: *IntPair*);
begin if *Count* ≥ *MaxIntPairSize* **then** *NatSetError*(*coOverflow*, 0);
if *Limit* = *Count* **then** *SetLimit*(*Limit* + *GrowLimit*(*Limit*));
Items↑[*Count*] ← *aItem*; *inc*(*Count*);
end;

346. Delete an element at an index.

procedure *IntPairSeq.AtDelete*(*aIndex* : *integer*);
  var *i*: *integer*;
  begin if $(aIndex < 0) \lor (aIndex \geq Count)$ **then**
    begin *NatSetError*(*coIndexError*, 0); *exit*; **end**;
  if $aIndex < Count - 1$ **then**
    for $i \leftarrow aIndex$ **to** $Count - 2$ **do** $Items\uparrow[i] \leftarrow Items\uparrow[i+1]$;
  Dec(*Count*);
  end;

347. Resizing an IntPair sequence.

procedure *IntPairSeq.SetLimit*(*aLimit* : *integer*);
  var *aItems*: *IntPairListPtr*;
  begin if $aLimit < Count$ **then** $aLimit \leftarrow Count$;
  if $aLimit > MaxIntPairSize$ **then** $ALimit \leftarrow MaxIntPairSize$;
  if $aLimit \neq Limit$ **then**
    begin if $ALimit = 0$ **then** $AItems \leftarrow$ **nil**
    else begin *GetMem*(*AItems*, *ALimit* ∗ *SizeOf*(*IntPair*));
      if $(Count \neq 0) \land (Items \neq$ **nil**$)$ **then** *Move*(*Items*↑, *aItems*↑, *Count* ∗ *SizeOf*(*IntPair*));
      end;
    if $Limit \neq 0$ **then** *FreeMem*(*Items*, *Limit* ∗ *SizeOf*(*IntPair*));
    $Items \leftarrow aItems$; $Limit \leftarrow aLimit$;
    end;
  end;

348. Deleting all entries. We just set the logical size to zero. It leaves everything else untouched.

procedure *IntPairSeq.DeleteAll*;
  begin $Count \leftarrow 0$; **end**;

349. Append a pair of integers. We create a new *IntPair* using X and Y, then append it to the caller.

procedure *IntPairSeq.AssignPair*(X, Y : *integer*);
  var *lIntPair*: *IntPair*;
  begin $lIntPair.X \leftarrow X$; $lIntPair.Y \leftarrow Y$; *Insert*(*lIntPair*);
  end;

Section 9.15. INT REL

350. This is used in the `iocorrel.pas`, `identify.pas`, `equalizer.pas`, the analyzer, and a polynomial library.

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡
```
  IntRelPtr = ↑IntRel;
  IntRel = object (IntPairSeq)
    constructor Init(aLimit : integer);
    procedure Insert(const aItem: IntPair); virtual;
    procedure AtInsert(aIndex : integer;
        const aItem: IntPair); virtual;
    function Search(X, Y : integer; var aIndex : integer): boolean; virtual;
    function IndexOf(X, Y : integer): integer;
    constructor CopyIntRel(var aFunc : IntRel);
    function IsMember(X, Y : integer): boolean; virtual;
    procedure AssignPair(X, Y : integer); virtual;
  end ;
```

351. Constructor. This is just the inherited constructor.

⟨ Int relation implementation 351 ⟩ ≡
```
{ IntRel }
constructor IntRel.Init(aLimit : integer);
  begin inherited Init(aLimit);
  end;
```
This code is used in section 183.

352. Inserting an entry.

```
procedure IntRel.Insert(const aItem: IntPair);
  var I: integer;
  begin if ¬Search(aItem.X, aItem.Y, I) then
    begin if (I < 0) ∨ (I > Count) then
      begin NatSetError(coIndexError, 0); exit; end;
    if Count ≥ MaxIntPairSize then NatSetError(coOverflow, 0);
        { Finished with the possible errors }
    if Limit = Count then SetLimit(Limit + GrowLimit(Limit));
    if I ≠ Count then Move(Items↑[I], Items↑[I + 1], (Count − I) ∗ SizeOf(IntPair));
    Items↑[I] ← aItem; inc(Count);
    end;

end;
```

353. Insert at a specific index.

procedure *IntRel.AtInsert*(*aIndex* : *integer*;
  const *aItem*: *IntPair*);
 begin if $(aIndex < 0) \lor (aIndex > Count)$ **then** *NatSetError*(*coIndexError*, *aIndex*);
 if $Count = Limit$ **then** *SetLimit*(*Limit* + *GrowLimit*(*Limit*));
   { Shift everything to the right by 1 }
 if $aIndex < Limit$ **then** *Move*(*Items*↑[*aIndex*], *Items*↑[*aIndex* + 1], (*Count* − *aIndex*) ∗ *SizeOf*(*IntPair*));
   { Update the items, increment the logical size }
 Items↑[*aIndex*] ← *aItem*; *inc*(*Count*);
 end;

354. Bisection search for a relation. Search through *IntRel* for a relation $X = Y$. Note that this is not symmetric, i.e., if we have $Y = X$ in the *IntRel*, then it will not match.

Mutates the *aIndex*. If the relation is missing, *aIndex* will return where it *should* be.

function *IntRel.Search*(*X*, *Y* : *integer*; **var** *aIndex* : *integer*): *boolean*;
 var *L*, *H*, *I*, *C*: *integer*;
 begin *Search* ← *False*; $L \leftarrow 0$; $H \leftarrow Count - 1$;
 while $L \le H$ **do**
  begin $I \leftarrow (L + H)$ **shr** 1; *C* ← *CompareIntPairs*(*Items*↑[*I*].*X*, *Items*↑[*I*].*Y*, *X*, *Y*);
  if $C < 0$ **then** $L \leftarrow I + 1$
  else begin $H \leftarrow I - 1$;
   if $C = 0$ **then**
    begin *Search* ← *True*; $L \leftarrow I$; **end**;
   end;
  end;
 aIndex ← *L*;
 end;

355. Copy constructor. This moves the contents of *aFunc* into the caller. It will mutate the caller *but not* the argument supplied. The *Move* function copies the contents of one region of memory to another.

constructor *IntRel.CopyIntRel*(**var** *aFunc* : *IntRel*);
 begin *Init*(*aFunc.Limit*); *Move*(*aFunc.Items*↑, *Items*↑, *aFunc.Limit* ∗ *SizeOf*(*IntPair*));
 Count ← *aFunc.Count*;
 end;

356. Index of a relation. This will return the index of the $X = Y$ entry. If it is absent from the caller, then return −1.

function *IntRel.IndexOf*(*X*, *Y* : *integer*): *integer*;
 var *I*: *integer*;
 begin *IndexOf* ← −1;
 if *Search*(*X*, *Y*, *I*) **then** *IndexOf* ← *I*;
 end;

357. Test for membership. This just tests if $X = Y$ is contained in the caller.

function *IntRel.IsMember*(*X*, *Y* : *integer*): *boolean*;
 var *I*: *integer*;
 begin *IsMember* ← *Search*(*X*, *Y*, *I*); **end**;

358.

procedure *IntRel.AssignPair*(*X*, *Y* : *integer*);
  begin if *IsMember*(*X*, *Y*) **then** *exit*;
  inherited AssignPair(*X*, *Y*);
  end;

Section 9.16. PARTIAL INTEGERS FUNCTIONS

359.

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡
  NatSetPtr = ↑*NatSet*;
  NatSet = **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 *IntersectWith*(**const** *fAnother*: *NatSet*);
      { ** }
    function *HasInDom*(*fElem* : *integer*): *boolean*; *virtual*;
    function *IsEqualTo*(**const** *fFunc*: *NatSet*): *boolean*;
    function *IsSubsetOf*(**const** *fFunc*: *NatSet*): *boolean*;
    function *IsSupersetOf*(**const** *fFunc*: *NatSet*): *boolean*;
    function *Misses*(**const** *fFunc*: *NatSet*): *boolean*;
    constructor *MoveNatSet*(**var** *fFunc* : *NatSet*);
  end ;

360. Constructor. The empty *NatSet* can be constructed with the usual initialization.

⟨ Partial integer function implementation 360 ⟩ ≡
    { Partial integers Functions }
constructor *NatSet*.*Init*(*aLimit*, *aDelta* : *integer*);
  begin *MObject*.*Init*; *Items* ← **nil**; *Count* ← 0; *Limit* ← 0; *Delta* ← *ADelta*; *SetLimit*(*ALimit*);
  Duplicates ← *False*;
  end;

This code is used in section 183.

361. Singleton constructor. This initializes the *Delta* set to 4, and the *aLimit* set to 0. Then insert the given integer.

constructor *NatSet*.*InitWithElement*(*X* : *integer*);
  begin *Init*(0, 4); *InsertElem*(*X*);
  end;

362. Destructor. This delegates the heavy work to *SetLimit*(0).

destructor *NatSet*.*Done*;
  begin *Count* ← 0; *SetLimit*(0);
  end;

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

```
procedure NatSet.Insert(const aItem: IntPair);
  var I: integer;
  begin if ¬SearchPair(aItem.X, I) ∨ Duplicates then
    begin if (I < 0) ∨ (I > Count) then   { Out of bounds, raise an error }
      begin NatSetError(coIndexError, 0); exit; end;
    if Limit = Count then   { Grow the capacity, if possible }
      begin if Delta = 0 then
        begin NatSetError(coOverFlow, 0); exit; end;
      SetLimit(Limit + Delta);
      end;
    if I ≠ Count then Move(Items↑[I], Items↑[I + 1], (Count − I) ∗ SizeOf(IntPair));
    Items↑[I] ← aItem; inc(Count);
    end;
  end;
```

364. Equality of IntPair objects. This just tests the componentwise equality of two *IntPair* objects.

```
function Equals(Key1, Key2 : IntPair): boolean;
  begin Equals ← (Key1.X = Key2.X) ∧ (Key1.Y = Key2.Y);
  end;
```

365. Search. This is a bisection search for any relation of the form $X = Y$ for some Y.

```
function NatSet.SearchPair(X : integer; var Index : integer): boolean;
  var L, H, I, C: integer;
  begin SearchPair ← False; L ← 0; H ← Count − 1;
  while L ≤ H do
    begin I ← (L + H) shr 1; C ← CompareInt(Items↑[I].X, X);
    if C < 0 then L ← I + 1
    else begin H ← I − 1;
      if C = 0 then
        begin SearchPair ← True;
        if ¬Duplicates then L ← I;
        end;
      end;
    end;
  Index ← L;
  end;
```

366. 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↑, Items↑, fFunc.Limit ∗ SizeOf(IntPair));
  Count ← fFunc.Count;
  end;
```

367. 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* ← *fFunc*; *fFunc.DeleteAll*; *fFunc.Limit* ← 0;
  fFunc.Items ← **nil**;
  end;

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

procedure *NatSet.EnlargeBy*(**const** *fAnother*: *NatSet*);
  var *I*: *integer*;
  begin for *I* ← 0 **to** *fAnother.Count* − 1 **do** *InsertElem*(*fAnother.Items*↑[*i*].*X*);
  end;

369. 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* ← 0 **to** *fAnother.Count* − 1 **do** *DeleteElem*(*fAnother.Items*↑[*i*].*X*);
  end;

370. Take intersection. This computes *Self* ← *Self* ∩ *Other*

procedure *NatSet.IntersectWith*(**const** *fAnother*: *NatSet*);
  var *k*: *integer*;
  begin *k* ← 0;
  while *k* < *Count* **do**
    if ¬*fAnother.HasInDom*(*Items*↑[*k*].*X*) **then** *AtDelete*(*k*)
    else *inc*(*k*);
  end;

371. Insert an element. We can insert $X = 0$ into the caller.

procedure *NatSet.InsertElem*(*X* : *integer*);
  var *lIntPair*: *IntPair*;
  begin *lIntPair.X* ← *X*; *lIntPair.Y* ← 0; *Insert*(*lIntPair*);
  end;

372. Deleting an element. Similarly, we can delete the first element of the form $X = Y$ for some Y.

procedure *NatSet.DeleteElem*(*fElem* : *integer*);
  var *I*: *integer*;
  begin if *SearchPair*(*fElem*, *I*) **then** *AtDelete*(*I*);
  end;

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

function *NatSet.HasInDom*(*fElem* : *integer*): *boolean*;
  var *I*: *integer*;
  begin *HasInDom* ← *SearchPair*(*fElem*, *I*);
  end;

374. 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* ← *false*;
  if *Count* ≠ *fFunc.Count* **then** *exit*;
  for $I \leftarrow 0$ **to** *Count* − 1 **do**
    if ¬*Equals*(*Items*↑[I], *fFunc.Items*↑[I]) **then** *exit*;
  IsEqualTo ← *true*;
  end;

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

function *NatSet.IsSubsetOf*(**const** *fFunc*: *NatSet*): *boolean*;
  var i, j, k, c: *integer*; { Jezeli sprawdzamy, czy mala funkcja jest zawarta w duzej, to to wykomentowane moze byc lepsze }
  begin *IsSubsetOf* ← *false*; c ← *fFunc.Count*;
  if $c <$ *Count* **then** *exit*;
  $j \leftarrow 0$;
  for $i \leftarrow 0$ **to** *Count* − 1 **do**
    begin k ← *Items*↑[i].X;
    while $(j < c) \land$ (*fFunc.Items*↑[j].$X < k$) **do** *inc*(j);
    if $(j = c) \lor$ ¬*Equals*(*fFunc.Items*↑[j], *Items*↑[i]) **then** *exit*;
    end;
  IsSubsetOf ← *true*;
  end;

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

function *NatSet.IsSupersetOf*(**const** *fFunc*: *NatSet*): *boolean*;
  begin *IsSupersetOf* ← *fFunc.IsSubsetOf*(*Self*);
  end;

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

function *NatSet.Misses*(**const** *fFunc*: *NatSet*): *boolean*;
  var I, k: *integer*;
  begin if *Count* > *fFunc.Count* **then**
    begin for $k \leftarrow 0$ **to** *fFunc.Count* − 1 **do**
      if *SearchPair*(*fFunc.Items*↑[k].X, I) **then**
        begin *Misses* ← *false*; *exit* **end**
    end
  else begin for $k \leftarrow 0$ **to** *Count* − 1 **do**
      if *fFunc.SearchPair*(*Items*↑[k].X, I) **then**
        begin *Misses* ← *false*; *exit* **end**;
    end;
  Misses ← *true*;
  end;

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

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

function *NatSet.ElemNr*(X : *integer*): *integer*;
  var I: *integer*;
  begin *ElemNr* $\leftarrow -1$;
  if *SearchPair*(X, I) **then** *ElemNr* $\leftarrow I$;
  end;

Section 9.17. FUNCTION OF NATURAL NUMBERS

379. The *NatFunc* is used in the analyzer, equalizer, unifier, and elsewhere. Its destructor is the only place where *nConsistent* ← *false*.

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡
```
NatFuncPtr = ↑NatFunc;
NatFunc = 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 CompareWith(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 ;
```

380. 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.

⟨ *NatFunc* implementation 380 ⟩ ≡
```
constructor NatFunc.InitNatFunc(ALimit, ADelta : integer);
  begin inherited Init(ALimit, ADelta); nConsistent ← true;
  end;
constructor NatFunc.CopyNatFunc(const fFunc: NatFunc);
  begin Init(fFunc.Limit, fFunc.Delta); Move(fFunc.Items↑, Items↑, fFunc.Limit * SizeOf(IntPair));
  Count ← fFunc.Count; nConsistent ← fFunc.nConsistent;
  end;
constructor NatFunc.MoveNatFunc(var fFunc : NatFunc);
  begin Init(fFunc.Limit, fFunc.Delta); Self ← fFunc; fFunc.DeleteAll; fFunc.Limit ← 0;
  fFunc.Items ← nil;
  end;
```

See also section 396.

This code is used in section 183.

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

For the ring (or ringoid) $\mathbf{N}^{\mathbf{N}}$, this amounts to

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

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

```
constructor NatFunc.LCM(const aFunc1, aFunc2: NatFunc);
  var i, j, m: integer;
  begin m ← aFunc2.Delta;
  if aFunc1.Delta > m then m ← aFunc1.Delta;
  InitNatFunc(aFunc1.Limit + aFunc2.Limit, m); i ← 0; j ← 0;
  while (i < aFunc1.Count) ∧ (j < aFunc2.Count) do
    case CompareInt(aFunc1.Items↑[i].X, aFunc2.Items↑[j].X) of
    −1: begin Insert(aFunc1.Items↑[i]); inc(i) end;
    0: begin   { m = max(f(i), g(i)) }
      m ← aFunc1.Items↑[i].Y;
      if aFunc2.Items↑[j].Y > m then m ← aFunc2.Items↑[j].Y;
      Assign(aFunc1.Items↑[i].X, m);   { destructively set f(i) ← m }
      inc(i); inc(j);
      end;
    1: begin Insert(aFunc2.Items↑[j]); inc(j) end;
    endcases;
  if i ≥ aFunc1.Count then
    for j ← j to aFunc2.Count − 1 do Insert(aFunc2.Items↑[j])
  else for i ← i to aFunc1.Count − 1 do Insert(aFunc1.Items↑[i]);
  end;
```

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

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

```
procedure NatFunc.Assign(X, Y : integer);
  var lIntPair: IntPair;
  begin if nConsistent then
    begin if HasInDom(X) ∧ (Value(X) ≠ Y) then
      begin Refuted; exit end;
    lIntPair.X ← X; lIntPair.Y ← Y; Insert(lIntPair);
    end;
  end;
```

383. Increment $f(x)$. Given a *NatFunc* object f, and an integer x, $f.Up(x)$ will

(1) If $x \in \mathrm{dom}(f)$, then update the value $f(x) \geq f(x) + 1$

(2) Otherwise, $x \notin \mathrm{dom}(f)$, 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*↑[*I*].*Y*)
    else begin *lIntPair*.*X* ← *X*; *lIntPair*.*Y* ← 1; *Insert*(*lIntPair*);
      end;
    end;
  end;

384. Decrement $f(x)$. Given a *NatFunc* object f, and an integer x, $f.Down(x)$ will

(1) If $x \in \mathrm{dom}(f)$, then update the value $f(x) \geq f(x) - 1$ and if this is then zero, remove it from the function.

(2) Otherwise, $x \notin \mathrm{dom}(f)$, so this corresponds to $f(x) = 0$, and we cannot mutate $f(x) \leftarrow -1$ without making it no longer natural-valued. So we raise an error.

procedure *NatFunc.Down*(*X* : *integer*);
  var *I*: *integer*;
  begin if *nConsistent* **then**
    begin if *SearchPair*(*X*, *I*) **then**
      begin *dec*(*Items*↑[*I*].*Y*);
      if *Items*↑[*I*].*Y* = 0 **then** *AtDelete*(*I*);
      end
    else *NatSetError*(*coConsistentError*, 0);
    end;
  end;

385. Getting the value of $f(x)$ when $x \in \mathrm{dom}(f)$. When $x \notin \mathrm{dom}(f)$, raise an error.

function *NatFunc.Value*(*fElem* : *integer*): *integer*;
  var *I*: *integer*;
  begin if *SearchPair*(*fElem*, *I*) **then** *Value* ← *Items*↑[*I*].*Y*
  else *NatSetError*(*coDuplicate*, 0);
  end;

386. Destructor. We usually try to extend partial functions on **N**, but if we end up trying to extend where it is already defined to a different value, then we arrive at an inconsistent extension. It is referred to as a "refuted" situation.

destructor *NatFunc.Refuted*;
  begin *inheritedDone*; *nConsistent* ← *false*
  end;

387. Join. For two partial functions $f\colon \mathbf{N} \rightharpoonup \mathbf{N}$ and $g\colon \mathbf{N} \rightharpoonup \mathbf{N}$, we form $f \cup g$ provided

$$f \cap g = f|_{\mathrm{dom}(f\cap g)} = g|_{\mathrm{dom}(f\cap g)}.$$

That is to say, for all $x \in \mathrm{dom}(f) \cap \mathrm{dom}(g)$, 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 ¬fFunc.nConsistent then
      begin Refuted; exit end;
    for k ← 0 to fFunc.Count − 1 do
      if SearchPair(fFunc.Items↑[k].X, I) then
        begin if ¬Equals(Items↑[I], fFunc.Items↑[k]) then
          begin Refuted; exit end;
        end
      else Insert(fFunc.Items↑[k]);
    end;
  end;
```

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

```
procedure NatFunc.EnlargeBy(fAnother : NatFuncPtr);   { ? virtual;? }
  var i, j, lCount, lLimit: integer; lItems: IntPairListPtr;
  begin if nConsistent then
    begin if ¬fAnother↑.nConsistent then
      begin Refuted; exit end;
    if fAnother↑.Count = 0 then exit;
    lCount ← Count; lItems ← Items; lLimit ← Limit; Limit ← 0; Count ← 0;
    SetLimit(lCount + fAnother↑.Count); i ← 0; j ← 0;
    while (i < lCount) ∧ (j < fAnother↑.Count) do
      case CompareInt(lItems↑[i].X, fAnother↑.Items↑[j].X) of
      −1: begin Insert(lItems↑[i]); inc(i) end;
      0: begin if Equals(lItems↑[i], fAnother↑.Items↑[j]) then Insert(lItems↑[i])
        else begin Refuted; FreeMem(lItems, lLimit ∗ SizeOf(IntPair)); exit end;
        inc(i); inc(j);
        end;
      1: begin Insert(fAnother↑.Items↑[j]); inc(j) end;
      endcases;
    if i ≥ lCount then
      for j ← j to fAnother↑.Count − 1 do Insert(fAnother↑.Items↑[j])
    else for i ← i to lCount − 1 do Insert(lItems↑[i]);
    SetLimit(0); FreeMem(lItems, lLimit ∗ SizeOf(IntPair));
    end;
  end;
```

389. We want to join two partial functions $f\colon \mathbf{N} \rightharpoonup \mathbf{N}$ and $g\colon \mathbf{N} \rightharpoonup \mathbf{N}$ 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* ← **nil**; *lEval.CopyNatFunc*(*Self*); *lEval.EnlargeBy*(*fLatAtom*);
  if *lEval.nConsistent* **then** *JoinAtom* ← *NatFuncPtr*(*lEval.CopyObject*);
  end;

390. Comparing partial functions. Given two partial functions, $f\colon \mathbf{N} \rightharpoonup \mathbf{N}$ and $g\colon \mathbf{N} \rightharpoonup \mathbf{N}$, 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 $x \in \operatorname{dom}(f)$, 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 $\operatorname{dom}(f)$, and then we return 0.

function *CompareNatFunc*(*aKey1*, *aKey2* : *Pointer*): *integer*;
  var *i*, *lInt*: *integer*;
  begin with *NatFuncPtr*(*aKey1*)↑ **do**
    begin *lInt* ← *CompareInt*(*Count*, *NatFuncPtr*(*aKey2*)↑.*Count*);
    if *lInt* ≠ 0 **then**
      begin *CompareNatFunc* ← *lInt*; *exit* **end**;
    for *i* ← 0 **to** *Count* − 1 **do**
      begin *lInt* ← *CompareInt*(*Items*↑[*i*].*X*, *NatFuncPtr*(*aKey2*)↑.*Items*↑[*i*].*X*);
      if *lInt* ≠ 0 **then**
        begin *CompareNatFunc* ← *lInt*; *exit* **end**;
      lInt ← *CompareInt*(*Items*↑[*i*].*Y*, *NatFuncPtr*(*aKey2*)↑.*Items*↑[*i*].*Y*);
      if *lInt* ≠ 0 **then**
        begin *CompareNatFunc* ← *lInt*; *exit* **end**;
      end;
    end;
  CompareNatFunc ← 0;
  end;

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

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

function *NatFunc.WeakerThan*(**const** *fNatFunc*: *NatFunc*): *boolean*;
  var *i*, *k*: *integer*;
  begin *WeakerThan* ← *false*;
  if *Count* ≤ *fNatFunc.Count* **then**
    begin for *k* ← 0 **to** *Count* − 1 **do**
      begin *i* ← *Items*↑[*k*].*X*;
      if ¬*fNatFunc.HasInDom*(*i*) **then** *exit*;
      if *Items*↑[*k*].*Y* ≠ *fNatFunc.Value*(*i*) **then** *exit*;
      end;
    WeakerThan ← *true*;
    end;
  end;

392. Let $f\colon \mathbf{N} \rightharpoonup \mathbf{N}$ and $g\colon \mathbf{N} \rightharpoonup \mathbf{N}$ be partial functions. We will say that f is a "multiple" of g if $\|g\| \leq \|f\|$ and for each $x \in \operatorname{dom}(g)$ we have $x \in \operatorname{dom}(f)$ and $g(x) \leq f(x)$.

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

```
function NatFunc.IsMultipleOf(const fNatFunc: NatFunc): boolean;
  var k, l: integer;
  begin IsMultipleOf ← false;
  if fNatFunc.Count ≤ Count then
    begin for k ← 0 to fNatFunc.Count − 1 do
      if ¬HasInDom(fNatFunc.Items↑[k].X) then exit
      else if Value(fNatFunc.Items↑[k].X) < fNatFunc.Items↑[k].Y then exit;
    IsMultipleOf ← true;
    end;
  end;
```

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

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

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

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

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

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

```
  { Uzywajac WeakerThan mozna skrocic CompareWith !!! }
function NatFunc.CompareWith(const fNatFunc: NatFunc): integer;
  var i, k: integer;
  begin CompareWith ← 0;
  if Count ≤ fNatFunc.Count then
    begin for k ← 0 to Count − 1 do
      begin i ← Items↑[k].X;
      if ¬fNatFunc.HasInDom(i) then exit;
      if Items↑[k].Y ≠ fNatFunc.Value(i) then exit;
      end;
    CompareWith ← −1; exit;
    end;
  if fNatFunc.Count ≤ Count then
    begin for k ← 0 to fNatFunc.Count − 1 do
      begin i ← fNatFunc.Items↑[k].X;
      if ¬HasInDom(i) then exit;
      if fNatFunc.Items↑[k].Y ≠ Value(i) then exit;
      end;
    CompareWith ← 1; exit;
    end;
  end;
```

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

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

$$g(x) \geq \mathit{High}(\mathit{integer}) - f(x)$$

for each $x \in \mathrm{dom}(f) \cap \mathrm{dom}(g)$.

```
procedure NatFunc.Add(const aFunc: NatFunc);
  var k, l: integer;
  begin l ← 0;
  for k ← 0 to aFunc.Count − 1 do
    begin while (l < Count) ∧ (Items↑[l].X < aFunc.Items↑[k].X) do inc(l);
    if (l < Count) ∧ (Items↑[l].X = aFunc.Items↑[k].X) then
      begin if ⟨Has overflow occurred in NatFunc.Add? 395⟩ then RunError(215);
      inc(Items↑[l].Y, aFunc.Items↑[k].Y);
      end
    else AtInsert(l, aFunc.Items↑[k]);
    end;
  end;
```

395. An overflow occurs if $f(x) + g(x)$ is greater than $\mathit{High}(\mathit{integer})$ (the maximum value for an integer).

```
⟨Has overflow occurred in NatFunc.Add? 395⟩ ≡
  Items↑[l].Y > (High(integer) − aFunc.Items↑[k].Y)
```

This code is used in section 394.

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

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

```
⟨NatFunc implementation 380⟩ +≡
function NatFunc.CountAll: integer;
  var k, l: integer;
  begin l ← 0;
  for k ← 0 to Count − 1 do inc(l, Items↑[k].Y);
  CountAll ← l;
  end;
```

Section 9.18. SEQUENCES OF NATURAL NUMBERS

397.

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡

```
NatSeq = object (NatFunc)
    constructor InitNatSeq(ALimit, ADelta : integer);
    procedure InsertElem(X : integer); virtual;
    function Value(fElem : integer): integer; virtual;
    function IndexOf(Y : integer): integer;
  end ;
```

398. Constructor.

⟨ NatSeq implementation 398 ⟩ ≡

```
constructor NatSeq.InitNatSeq(ALimit, ADelta : integer);
  begin inherited Init(ALimit, ADelta); nConsistent ← true;
  end;
```

This code is used in section 183.

399. If we have a finite sequence $(a_0, \ldots, a_{n-1})$, then inserting an element x into it will yield the finite sequence $(a_0, \ldots, a_{n-1}, x)$.

```
procedure NatSeq.InsertElem(X : integer);
  var lPair: IntPair;
  begin lPair.X ← Count; lPair.Y ← X; inherited Insert(lPair);
  end;
```

400. The value for the k^{th} element in a sequence $(a_0, \ldots, a_{n-1})$ is a_k when $0 \le k < n$, and we take it to be 0 otherwise.

```
function NatSeq.Value(fElem : integer): integer;
    begin
    if   { (0¡=ind) and }
    (fElem < count) then Value ← Items↑[fElem].Y
  else Value ← 0;
    end ;
```

401. The index for a_i in the sequence $(a_0, \ldots, a_{n-1})$ is i when a_i is in the sequence. Otherwise, we return -1.

```
function NatSeq.IndexOf(Y : integer): integer;
  var lResult: integer;
  begin for lResult ← Count − 1 downto 0 do
    if Items↑[lResult].Y = Y then
      begin IndexOf ← lResult; exit
      end;
  IndexOf ← −1;
  end;
```

Section 9.19. INTEGER SEQUENCES

402.

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡
  IntegerListPtr = ↑*IntegerList*;
  IntegerList = **array** [0 .. *MaxIntegerListSize* − 1] **of** *integer*;
  PIntSequence = ↑*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 ;

403. We will need to quicksort lists of integers. This will mutate the *aList* argument, making it sorted. See also §257 and §334.

This procedure does not appear to be used anywhere in Mizar.

⟨ *IntSequence* implementation 403 ⟩ ≡
    { integer Sequences & Sets }
procedure *IntQuickSort*(*aList* : *IntegerListPtr*; *L*, *R* : *integer*);
  var *I*, *J*, *P*, *lTemp*: *integer*;
  begin repeat $I \leftarrow L$; $J \leftarrow R$; $P \leftarrow aList\uparrow[(L + R)$ **shr** $1]$;
    repeat while *CompareInt*(*aList*↑[*I*], *P*) < 0 **do** *inc*(*I*);
      while *CompareInt*(*aList*↑[*J*], *P*) > 0 **do** *Dec*(*J*);
      if $I \le J$ **then**
        begin $lTemp \leftarrow aList\uparrow[I]$; $aList\uparrow[I] \leftarrow aList\uparrow[J]$; $aList\uparrow[J] \leftarrow lTemp$; *inc*(*I*); *Dec*(*J*);
        end;
    until $I > J$;
    if $L < J$ **then** *IntQuickSort*(*aList*, *L*, *J*);
    $L \leftarrow I$;
  until $I \ge R$;
  end;

This code is used in section 183.

404. Constructor. We can create an empty sequence of integers, with a given capacity.

constructor *IntSequence.Init*(*aCapacity* : *integer*);
  begin *inherited Init*; *fList* ← **nil**; *fCount* ← 0; *fCapacity* ← 0; *SetCapacity*(*aCapacity*);
  end;

405. Copy constructor. We can copy an existing sequence.

constructor *IntSequence.CopySequence*(**const** *aSeq*: *IntSequence*);
  begin *Init*(*aSeq.fCapacity*); *AddSequence*(*aSeq*);
  end;

406. Move constructor. We can create a new array in heap, and move all the elements from a given sequence over, then free up the given sequence.

constructor *IntSequence.MoveSequence*(**var** *aSeq* : *IntSequence*);
  begin *inherited Init*; *fCount* ← *aSeq.fCount*; *fCapacity* ← *aSeq.fCapacity*; *fList* ← *aSeq.fList*;
  aSeq.fCount ← 0; *aSeq.fCapacity* ← 0; *aSeq.fList* ← **nil**;
  end;

407. Destructor.

destructor *IntSequence.Done*;
  begin *inherited Done*; *fCount* ← 0; *SetCapacity*(0);
  end;

408. 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↑[*fCount*] ← *aInt*; *Insert* ← *fCount*; *inc*(*fCount*);
  end;

409. Appending a sequence. This takes a finite sequence $(a_0, \ldots, a_{n-1})$ and another finite sequence $(b_0, \ldots, b_{m-1})$, then forms a new finite sequence $(a_0, \ldots, a_{n-1}, b_0, \ldots, b_{m-1})$. It mutates the caller.

procedure *IntSequence.AddSequence*(**const** *aSeq*: *IntSequence*);
  var I, r: *integer*;
  begin for I ← 0 **to** *aSeq.fCount* − 1 **do** r ← *Insert*(*aSeq.fList*↑[I]);
  end;

410. Clearing a sequence.

procedure *IntSequence.Clear*;
  begin if *fCount* ≠ 0 **then**
    begin *fCount* ← 0; *SetCapacity*(0);
    end;
  end;

411. Delete entry in sequence. Removing the i^{th} entry in the sequence $(a_0, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_{n-1})$ yields the finite sequence $(a_0, \ldots, a_{i-1}, a_{i+1}, \ldots, a_{n-1})$. If $i < 0$ or $n - 1 < i$, then we raise an error.

procedure *IntSequence.AtDelete*(*aIndex* : *integer*);
  begin if (*aIndex* < 0) ∨ (*aIndex* ≥ *fCount*) **then** *IntListError*(*coIndexError*, *aIndex*);
  Dec(*fCount*);
  if *aIndex* < *fCount* **then** *Move*(*fList*↑[*aIndex* + 1], *fList*↑[*aIndex*], (*fCount* − *aIndex*) ∗ *SizeOf*(*integer*));
  end;

412. We report errors using this helper function.

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

413. Let $(a_0, \ldots, a_{n-1})$ be a finite sequence. The value at index i is a_i when $0 \le i \le n-1$, otherwise it raises an error.

function *IntSequence.Value*(*aIndex* : *integer*): *integer*;
  begin if $(aIndex < 0) \lor (aIndex \ge fCount)$ **then** *IntListError*(*coIndexError*, *aIndex*);
  Value ← *fList*↑[*aIndex*];
  end;

414. For a finite sequence $(a_0, \ldots, a_{n-1})$ and a value x, if there is some entry $a_i = x$ with $a_j \ne x$ for $j < i$, then return i. Otherwise return -1.

function *IntSequence.IndexOf*(*aInt* : *integer*): *integer*;
  var *lResult*: *integer*;
  begin for *lResult* ← *fCount* − 1 **downto** 0 **do**
    if *fList*↑[*lResult*] = *aInt* **then**
      begin *IndexOf* ← *lResult*; *exit*
      end;
  IndexOf ← −1;
  end;

415. Given a finite sequence $(a_0, \ldots, a_{n-1})$, an index i, and a value x:
(1) If $i < 0$ or i is too big, raise an error.
(2) If the logical size of the sequence equals its capacity, then grow the underlying array.
(3) If i is less than the logical size $i < n - 1$, then shift all the entries to the right by 1 so we have $(a_0, \ldots, a_{i-1}, 0, a_i, \ldots, a_{n-1})$
(4) Set the i^{th} entry to x, so we end up with the caller becoming $(a_0, \ldots, a_{i-1}, x, a_i, \ldots, a_{n-1})$.

procedure *IntSequence.AtInsert*(*aIndex*, *aInt* : *integer*);
  begin if $(aIndex < 0) \lor (aIndex > fCount)$ **then** *IntListError*(*coIndexError*, *aIndex*);
  if *fCount* = *fCapacity* **then** *SetCapacity*(*fCapacity* + *GrowLimit*(*fCapacity*));
  if *aIndex* < *fCount* **then** *Move*(*fList*↑[*aIndex*], *fList*↑[*aIndex* + 1], (*fCount* − *aIndex*) ∗ *SizeOf*(*integer*));
  fList↑[*aIndex*] ← *aInt*; *inc*(*fCount*);
  end;

416. Update entry of sequence. For a sequence $(a_0, \ldots, a_{n-1})$, an index i, and a new value x, if $0 \le i \le n-1$ then we set $a_i \leftarrow x$. Otherwise we have the index be out of bounds ($0 < i$ 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↑[*aIndex*] ← *aInt*;
  end;

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

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

Section 9.20. INTEGER SETS

418.

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡
  PIntSet = ↑*IntSet*;
  IntSetPtr = *pIntSet*;
  IntSet = **object** (*IntSequence*)
    function *Insert*(*aInt* : *integer*): *integer*; *virtual*;
    function *DeleteInt*(*aInt* : *integer*): *integer*; *virtual*;
    function *Find*(*aInt* : *integer*; **var** *aIndex* : *integer*): *boolean*; *virtual*;
    function *IndexOf*(*aInt* : *integer*): *integer*; *virtual*;
    procedure *AtInsert*(*aIndex*, *aInt* : *integer*); *virtual*;
    function *IsInSet*(*aInt* : *integer*): *boolean*; *virtual*;
    function *IsEqualTo*(**const** *aSet*: *IntSet*): *boolean*; *virtual*;
    function *IsSubsetOf*(**const** *aSet*: *IntSet*): *boolean*; *virtual*;
    function *IsSupersetOf*(**var** *aSet* : *IntSet*): *boolean*; *virtual*;
    function *Misses*(**var** *aSet* : *IntSet*): *boolean*; *virtual*;
  end ;

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

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

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

This code is used in section 183.

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

function *IntSet.DeleteInt*(*aInt* : *integer*): *integer*;
  var *lIndex*: *integer*;
  begin *DeleteInt* ← −1;
  if *Find*(*aInt*, *lIndex*) **then**
    begin *DeleteInt* ← *lIndex*; *AtDelete*(*lIndex*) **end**
  end;

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

function *IntSet.Find*(*aInt* : *integer*; **var** *aIndex* : *integer*): *boolean*;
  var L, H, I, C: *integer*;
  begin *Find* ← *False*; $L \leftarrow 0$; $H \leftarrow fCount - 1$;
  while $L \le H$ **do**
    begin $I \leftarrow (L + H)$ **shr** 1; $C \leftarrow$ *CompareInt*(*fList*↑[I], *aInt*);
    if $C < 0$ **then** $L \leftarrow I + 1$
    else begin $H \leftarrow I - 1$;
      if $C = 0$ **then**
        begin *Find* ← *True*; $L \leftarrow I$; **end**;
      end;
    end;
  aIndex ← L;
  end;

422. 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 ¬*Find*(*aInt*, *lResult*) **then** *lResult* ← −1;
  IndexOf ← *lResult*;
  end;

423. 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;

424. 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* ← *Find*(*aInt*, I);
  end;

425. Testing if two finite sets A and B of integers are equal requires $|A| = |B|$ and for each $x \in A$ we have $x \in B$. If these conditions are not both met, then $A \neq B$.

function *IntSet.IsEqualTo*(**const** *aSet*: *IntSet*): *boolean*;
  var I: *integer*;
  begin *IsEqualTo* ← *false*;
  if *fCount* ≠ *aSet.fCount* **then** *exit*;
  for $I \leftarrow 0$ **to** $fCount - 1$ **do**
    if *fList*↑[I] ≠ *aSet.fList*↑[I] **then** *exit*;
  IsEqualTo ← *true*;
  end;

426. Subset predicate. We can test $A \subseteq B$ by $|A| \le |B|$ and for each $a \in A$ we have $a \in B$.

```
function IntSet.IsSubsetOf(const aSet: IntSet): boolean;
  var i, j, lInt: integer;
  begin IsSubsetOf ← false;
  if aSet.fCount < fCount then exit;
  j ← 0;  { index of B }
  for i ← 0 to fCount − 1 do  { loop over a ∈ A }
    begin lInt ← fList↑[i];
    while (j < aSet.fCount) ∧ (aSet.fList↑[j] < lInt) do inc(j);
    if (j = aSet.fCount) ∨ (aSet.fList↑[j] ≠ fList↑[i]) then exit;
    end;
  IsSubsetOf ← true;
  end;
```

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

```
function IntSet.IsSupersetOf(var aSet : IntSet): boolean;
  begin IsSupersetOf ← aSet.IsSubsetOf(Self);
  end;
```

428. Test for disjointness. We have $A \cap B = \emptyset$ if every $a \in A$ is such that $a \notin B$.

```
function IntSet.Misses(var aSet : IntSet): boolean;
  var k: integer;
  begin if fCount > aSet.fCount then
    begin for k ← 0 to aSet.fCount − 1 do
      if IsInSet(aSet.fList↑[k]) then
        begin Misses ← false; exit end
    end
  else begin for k ← 0 to fCount − 1 do
      if aSet.IsInSet(fList↑[k]) then
        begin Misses ← false; exit end;
    end;
  Misses ← true;
  end;
```

Section 9.21. PARTIAL BINARY INTEGER FUNCTIONS

429. We want to describe partial functions like $f\colon \mathbf{Z} \times \mathbf{Z} \rightharpoonup \mathbf{Z}$. 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.

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡
IntTripletListPtr = ↑*IntTripletList*;
IntTripletList = **array** [0 . . *MaxIntTripletSize* − 1] **of** *IntTriplet*;
BinIntFuncPtr = ↑*BinIntFunc*;
BinIntFunc = **object** (*MObject*)
fList: *IntTripletListPtr*;
fCount: *integer*;
fCapacity: *integer*;
constructor *Init*(*aLimit* : *integer*);
procedure *BinIntFuncError*(*aCode*, *aInfo* : *integer*); *virtual*;
destructor *Done*; *virtual*;
procedure *Insert*(**const** *aItem*: *IntTriplet*); *virtual*;
procedure *AtDelete*(*aIndex* : *integer*);
procedure *SetCapacity*(*aLimit* : *integer*); *virtual*;
procedure *DeleteAll*;
function *Search*(*X1*, *X2* : *integer*; **var** *aIndex* : *integer*): *boolean*; *virtual*;
function *IndexOf*(*X1*, *X2* : *integer*): *integer*;
constructor *CopyBinIntFunc*(**var** *aFunc* : *BinIntFunc*);
function *HasInDom*(*X1*, *X2* : *integer*): *boolean*; *virtual*;
procedure *Assign*(*X1*, *X2*, *Y* : *integer*); *virtual*;
procedure *Up*(*X1*, *X2* : *integer*); *virtual*;
procedure *Down*(*X1*, *X2* : *integer*); *virtual*;
function *Value*(*X1*, *X2* : *integer*): *integer*; *virtual*;
procedure *Add*(**const** *aFunc*: *BinIntFunc*); *virtual*;
function *CountAll*: *integer*; *virtual*;
end ;

430. We have a convenience function for reporting errors.

⟨ Partial Binary integer Functions 430 ⟩ ≡
procedure *BinIntFunc*.*BinIntFuncError*(*aCode*, *aInfo* : *integer*);
begin *RunError*(212 − *aCode*); **end**;

This code is used in section 183.

431. Constructor. We initialize the empty partial function.

constructor *BinIntFunc*.*Init*(*aLimit* : *integer*);
begin *MObject*.*Init*; *fList* ← **nil**; *fCount* ← 0; *fCapacity* ← 0; *SetCapacity*(*aLimit*);
end;

432. Destructor.

destructor *BinIntFunc*.*Done*;
begin *fCount* ← 0; *SetCapacity*(0);
end;

433. If we have a partial function $f\colon \mathbf{Z}\times\mathbf{Z}\rightharpoonup\mathbf{Z}$ and a triple (x,y,z), then check if $(x,y)\in\mathrm{dom}(f)$. If so, we're done.

Otherwise we add $f(x,y)=z$ to the partial function.

```
procedure BinIntFunc.Insert(const aItem: IntTriplet);
  var I: integer;
  begin if ¬Search(aItem.X1, aItem.X2, I) then
    begin if (I < 0) ∨ (I > fCount) then
      begin BinIntFuncError(coIndexError, 0); exit; end;
    if fCapacity = fCount then SetCapacity(fCapacity + GrowLimit(fCapacity));
    if I ≠ fCount then Move(fList↑[I], fList↑[I + 1], (fCount − I) ∗ SizeOf(IntTriplet));
    fList↑[I] ← aItem; inc(fCount);
    end;
  end;
```

434. Given $f\colon \mathbf{Z}\times\mathbf{Z}\rightharpoonup\mathbf{Z}$, we represent it as an array of $\mathbf{Z}\times\mathbf{Z}\times\mathbf{Z}$. So we can remove the entry at index i when $0\leq i<\|f\|$. Otherwise when $i<0$ or $\|f\|\leq i$, raise an error.

```
procedure BinIntFunc.AtDelete(aIndex : integer);
  var i: integer;
  begin if (aIndex < 0) ∨ (aIndex ≥ fCount) then
    begin BinIntFuncError(coIndexError, 0); exit; end;
  if aIndex < fCount − 1 then
    for i ← aIndex to fCount − 2 do fList↑[i] ← fList↑[i + 1];
  Dec(fCount);
  end;
```

435. Ensure capacity.

```
procedure BinIntFunc.SetCapacity(aLimit : integer);
  var aItems: IntTripletListPtr;
  begin if aLimit < fCount then aLimit ← fCount;
  if aLimit > MaxIntTripletSize then ALimit ← MaxIntTripletSize;
  if aLimit ≠ fCapacity then
    begin if ALimit = 0 then AItems ← nil
    else begin GetMem(AItems, ALimit ∗ SizeOf(IntTriplet));
      if (fCount ≠ 0) ∧ (fList ≠ nil) then Move(fList↑, aItems↑, fCount ∗ SizeOf(IntTriplet));
      end;
    if fCapacity ≠ 0 then FreeMem(fList, fCapacity ∗ SizeOf(IntTriplet));
    fList ← aItems; fCapacity ← aLimit;
    end;
  end;
```

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

```
procedure BinIntFunc.DeleteAll;
  begin fCount ← 0; end;
```

437. We can use bisection search to find an entry (x_1, x_2) such that $(x_1, x_2) \in \mathrm{dom}(f)$.

```
function BinIntFunc.Search(X1, X2 : integer; var aIndex : integer): boolean;
  var L, H, I, C: integer;
  begin Search ← False; L ← 0; H ← fCount − 1;
  while L ≤ H do
    begin I ← (L + H) shr 1; C ← CompareIntPairs(fList↑[I].X1, fList↑[I].X2, X1, X2);
    if C < 0 then L ← I + 1
    else begin H ← I − 1;
      if C = 0 then
        begin Search ← True; L ← I; end;
      end;
    end;
  aIndex ← L;
  end;
```

438. Copy constructor. This leaves *aFunc* unchanged, and clones *aFunc*.

```
constructor BinIntFunc.CopyBinIntFunc(var aFunc : BinIntFunc);
  begin Init(aFunc.fCapacity); Move(aFunc.fList↑, fList↑, aFunc.fCapacity ∗ SizeOf(IntTriplet));
  fCount ← aFunc.fCount;
  end;
```

439. Given $f: \mathbf{Z} \times \mathbf{Z} \rightharpoonup \mathbf{Z}$ and (x_1, x_2), find the index for the underlying dynamic array i such that it contains $(x_1, x_2, f(x_1, x_2))$. If there is no such entry, $i = -1$ is returned.

```
function BinIntFunc.IndexOf(X1, X2 : integer): integer;
  var I: integer;
  begin IndexOf ← −1;
  if Search(X1, X2, I) then IndexOf ← I;
  end;
```

440. Test if $(x_1, x_2) \in \mathrm{dom}(f)$.

```
function BinIntFunc.HasInDom(X1, X2 : integer): boolean;
  var I: integer;
  begin HasInDom ← Search(X1, X2, I);
  end;
```

441. Given $f: \mathbf{Z} \times \mathbf{Z} \rightharpoonup \mathbf{Z}$, and $(x_1, x_2) \in \mathbf{Z} \times \mathbf{Z}$ and $y \in \mathbf{Z}$, try setting $f(x_1, x_2) = y$ provided $(x_1, x_2) \notin \mathrm{dom}(f)$ 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) ∧ (Value(X1, X2) ≠ Y) then
    begin BinIntFuncError(coDuplicate, 0); exit
    end;
  lIntTriplet.X1 ← X1; lIntTriplet.X2 ← X2; lIntTriplet.Y ← Y; Insert(lIntTriplet);
  end;
```

442. Given $f\colon \mathbf{Z}\times\mathbf{Z} \rightharpoonup \mathbf{Z}$ and $(x_1,x_2)\in\mathbf{Z}\times\mathbf{Z}$. If $(x_1,x_2)\in\operatorname{dom}(f)$, 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*↑[*I*].*Y*)
  else begin *lIntTriplet.X1* ← *X1*; *lIntTriplet.X2* ← *X2*; *lIntTriplet.Y* ← 1; *Insert*(*lIntTriplet*);
    end;
  end;

443. Given $f\colon \mathbf{Z}\times\mathbf{Z} \rightharpoonup \mathbf{Z}$ and $(x_1,x_2)\in\mathbf{Z}\times\mathbf{Z}$. If $(x_1,x_2)\in\operatorname{dom}(f)$, 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 $(x_1,x_2)\notin\operatorname{dom}(f)$, raise an error.

procedure *BinIntFunc.Down*(*X1*, *X2* : *integer*);
  var *I*: *integer*;
  begin if *Search*(*X1*, *X2*, *I*) **then**
    begin *dec*(*fList*↑[*I*].*Y*);
    if *fList*↑[*I*].*Y* = 0 **then** *AtDelete*(*I*);
    end
  else *BinIntFuncError*(*coConsistentError*, 0);
  end;

444. Given $f\colon \mathbf{Z}\times\mathbf{Z} \rightharpoonup \mathbf{Z}$, and $(x_1,x_2)\in\mathbf{Z}\times\mathbf{Z}$, if $(x_1,x_2)\notin\operatorname{dom}(f)$ then raise an error. Otherwise when $(x_1,x_2)\in\operatorname{dom}(f)$, return $f(x_1,x_2)$.

function *BinIntFunc.Value*(*X1*, *X2* : *integer*): *integer*;
  var *I*: *integer*;
  begin if *Search*(*X1*, *X2*, *I*) **then** *Value* ← *fList*↑[*I*].*Y*
  else *BinIntFuncError*(*coDuplicate*, 0);
  end;

445. Given two partial functions $f,g\colon \mathbf{Z}\times\mathbf{Z} \rightharpoonup \mathbf{Z}$, compute $f+g\colon \mathbf{Z}\times\mathbf{Z} \rightharpoonup \mathbf{Z}$. This is defines by:
(1) For $(x_1,x_2)\in\operatorname{dom}(f)\cap\operatorname{dom}(g)$, set $(f+g)(x_1,x_2)=f(x_1,x_2)+g(x_1,x_2)$
(2) For $(x_1,x_2)\in\operatorname{dom}(f)\setminus\operatorname{dom}(g)$, set $(f+g)(x_1,x_2)=f(x_1,x_2)$
(3) For $(x_1,x_2)\in\operatorname{dom}(g)\setminus\operatorname{dom}(f)$, 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* ← 0 **to** *aFunc.fCount* − 1 **do**
    if *Search*(*aFunc.fList*↑[*k*].*X1*, *aFunc.fList*↑[*k*].*X2*, *l*) **then** *inc*(*fList*↑[*l*].*Y*, *aFunc.fList*↑[*k*].*Y*)
    else *Assign*(*aFunc.fList*↑[*k*].*X1*, *aFunc.fList*↑[*k*].*X2*, *aFunc.fList*↑[*k*].*Y*);
  end;

446. Sum. For $f\colon \mathbf{Z}\times\mathbf{Z} \rightharpoonup \mathbf{Z}$, we compute

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

function *BinIntFunc.CountAll*: *integer*;
  var *k*, *l*: *integer*;
  begin *l* ← 0;
  for *k* ← 0 **to** *fCount* − 1 **do** *inc*(*l*, *fList*↑[*k*].*Y*);
  CountAll ← *l*;
  end;

Section 9.22. PARTIAL INTEGERS TO PAIR OF INTEGERS FUNCTIONS

447.

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡
  Int2PairOfInt = **record** *X*, *Y1*, *Y2*: *integer*;
    end;
  Int2PairOfIntFuncPtr = ↑*Int2PairOfIntFunc*;
  Int2PairOfIntFunc = **object** (*MObject*)
    fList: **array of** *Int2PairOfInt*;
    fCount: *integer*;
    fCapacity: *integer*;
    constructor *Init*(*aLimit* : *integer*);
    procedure *Int2PairOfIntFuncError*(*aCode*, *aInfo* : *integer*); *virtual*;
    destructor *Done*; *virtual*;
    procedure *Insert*(**const** *aItem*: *Int2PairOfInt*); *virtual*;
    procedure *AtDelete*(*aIndex* : *integer*);
    procedure *SetCapacity*(*aLimit* : *integer*); *virtual*;
    procedure *DeleteAll*;
    function *Search*(*X* : *integer*; **var** *aIndex* : *integer*): *boolean*; *virtual*;
    function *IndexOf*(*X* : *integer*): *integer*;
    constructor *CopyInt2PairOfIntFunc*(**var** *aFunc* : *Int2PairOfIntFunc*);
    function *HasInDom*(*X* : *integer*): *boolean*; *virtual*;
    procedure *Assign*(*X*, *Y1*, *Y2* : *integer*); *virtual*;
    function *Value*(*X* : *integer*): *IntPair*; *virtual*;
  end ;

448. We have a helper function for raising errors.

⟨ Partial integers to Pair of integers Functions 448 ⟩ ≡
    { Partial integers to Pair of integers Functions }
procedure *Int2PairOfIntFunc*.*Int2PairOfIntFuncError*(*aCode*, *aInfo* : *integer*);
  begin *RunError*(212 − *aCode*);
  end;

This code is used in section 183.

449. Constructor. Creates an empty $f\colon \mathbf{Z} \rightharpoonup \mathbf{Z}\times\mathbf{Z}$ with an underlying dynamic array whose capacity is given as the argument *aLimit*.

constructor *Int2PairOfIntFunc*.*Init*(*aLimit* : *integer*);
  begin *MObject*.*Init*; *fList* ← **nil**; *fCount* ← 0; *fCapacity* ← 0; *SetCapacity*(*aLimit*);
  end;

450. Destructor.

destructor *Int2PairOfIntFunc*.*Done*;
  begin *fCount* ← 0; *SetCapacity*(0);
  end;

451. Inserting (x, y_1, y_2) into $f\colon \mathbf{Z} \rightharpoonup \mathbf{Z}\times\mathbf{Z}$ amounts to checking if $(x, y_1, y_2)\in f$. If not, then insert the entry.

Otherwise, if $(x, y_1, y_2)\notin f$ but $x\in\mathrm{dom}(f)$, then raise an error.

Otherwise do nothing.

procedure *Int2PairOfIntFunc.Insert*(**const** *aItem*: *Int2PairOfInt*);
  var *I*: *integer*;
  begin if $\neg$*Search*(*aItem.X*, *I*) **then**
    begin if $(I < 0) \lor (I > fCount)$ **then**
      begin *Int2PairOfIntFuncError*(*coIndexError*, 0); *exit*; **end**;
    if *fCapacity* = *fCount* **then** *SetCapacity*(*fCapacity* + *GrowLimit*(*fCapacity*));
    if $I \neq fCount$ **then** *Move*(*fList*[*I*], *fList*[*I* + 1], (*fCount* − *I*) ∗ *SizeOf*(*Int2PairOfInt*));
    fList[*I*] ← *aItem*; *inc*(*fCount*);
    end
  else if (*fList*[*I*].*Y1* ≠ *aItem*.*Y1*) ∨ (*fList*[*I*].*Y2* ≠ *aItem*.*Y2*) **then**
      begin *Int2PairOfIntFuncError*(*coDuplicate*, 0); *exit*; **end**;
  end;

452. Delete an entry from the underlying dynamic array. Raise an error if the index given is out of bounds.

procedure *Int2PairOfIntFunc.AtDelete*(*aIndex* : *integer*);
  var *i*: *integer*;
  begin if $(aIndex < 0) \lor (aIndex \geq fCount)$ **then**
    begin *Int2PairOfIntFuncError*(*coIndexError*, 0); *exit*;
    end;
  if $aIndex < fCount - 1$ **then**
    for $i \leftarrow aIndex$ **to** $fCount - 2$ **do** *fList*[*i*] ← *fList*[*i* + 1];
  dec(*fCount*);
  end;

453.

procedure *Int2PairOfIntFunc.SetCapacity*(*aLimit* : *integer*);
  begin if $aLimit < fCount$ **then** $aLimit \leftarrow fCount$;
  setlength(*fList*, *aLimit*); *fCapacity* ← *aLimit*;
  end;

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

procedure *Int2PairOfIntFunc.DeleteAll*;
  begin *fCount* ← 0;
  end;

455. We can bisection search on the domain.

```
function Int2PairOfIntFunc.Search(X : integer; var aIndex : integer): boolean;
  var L, H, I, C: integer;
  begin Search ← False; L ← 0; H ← fCount − 1;
  while L ≤ H do
    begin I ← (L + H) shr 1; C ← CompareInt(fList[I].X, X);
    if C < 0 then L ← I + 1
    else begin H ← I − 1;
      if C = 0 then
        begin Search ← True; L ← I;
        end;
      end;
    end;
  aIndex ← L;
  end;
```

456. Copy constructor. This leaves the argument *aFunc* unchanged.

```
constructor Int2PairOfIntFunc.CopyInt2PairOfIntFunc(var aFunc : Int2PairOfIntFunc);
  begin Init(aFunc.fCapacity); Move(aFunc.fList[0], fList[0], aFunc.fCapacity ∗ SizeOf(Int2PairOfInt));
  fCount ← aFunc.fCount;
  end;
```

457. Find the index in the underlying dynamic array for $x \in \mathrm{dom}(f)$. If $x \notin \mathrm{dom}(f)$, then return -1.

```
function Int2PairOfIntFunc.IndexOf(X : integer): integer;
  var I: integer;
  begin IndexOf ← −1;
  if Search(X, I) then IndexOf ← I;
  end;
```

458. Test if $x \in \mathrm{dom}(f)$.

```
function Int2PairOfIntFunc.HasInDom(X : integer): boolean;
  var I: integer;
  begin HasInDom ← Search(X, I);
  end;
```

459. Attempt to insert (x, y_1, y_2) into $f\colon \mathbf{Z} \rightharpoonup \mathbf{Z} \times \mathbf{Z}$.

```
procedure Int2PairOfIntFunc.Assign(X, Y1, Y2 : integer);
  var lInt2PairOfInt: Int2PairOfInt;
  begin lInt2PairOfInt.X ← X; lInt2PairOfInt.Y1 ← Y1; lInt2PairOfInt.Y2 ← Y2;
  Insert(lInt2PairOfInt);
  end;
```

460. Given $f\colon \mathbf{Z} \rightharpoonup \mathbf{Z} \times \mathbf{Z}$ and $x \in \mathbf{Z}$, if $x \in \mathrm{dom}(f)$ 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 ← fList[I].Y1; Result.Y ← fList[I].Y2;
    end
  else Int2PairOfIntFuncError(coDuplicate, 0);
  end;
```

461. We have a myriad of random declarations, so we just stick them all here.

⟨ Public interface for `mobjects.pas` 184 ⟩ +≡
  { Comparing Strings wrt MStrObj }
function *CompareStringPtr*(*aKey1* , *aKey2* : *Pointer*): *integer*;

  { Comparing Strings and integers }
function *CompareStr*(*aStr1* , *aStr2* : *String*): *integer*;
function *CompareIntPairs*(*X1* , *Y1* , *X2* , *Y2* : *Longint*): *integer*;

  { Dynamic String handling routines }
function *NewStr*(**const** *S*: *String*): *PString*;
procedure *DisposeStr*(*P* : *PString*);

function *GrowLimit*(*aLimit* : *integer*): *integer*; { Abstract notification procedure }
function *CompareNatFunc*(*aKey1* , *aKey2* : *Pointer*): *integer*;
procedure *Abstract1* ;
var *EmptyNatFunc*: *NatFunc*;

File 10

XML Dictionary

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

⟨ xml_dict.pas 462 ⟩ ≡
⟨ GNU License 4 ⟩

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

File 11

Environment library

463. We have a library to handle accessing the Mizar mathematical library files. This is used in `makeenv.dpr` and using local `prel/` directories.

This will execute *InitLibrEnv* (§482) and *CheckCompatibility* (§479).

⟨ librenv.pas 463 ⟩ ≡
  ⟨ GNU License 4 ⟩
unit *librenv*;
  interface
  uses *mobjects*;
    ⟨ Interface for `MIZFILES` library 464 ⟩
  implementation
  uses
    `@{@&$`*IFDEF WIN32* `@}`*windows*,
    `@{@&$`*ENDIF* `@}`*mizenv*, *pcmizver*, *mconsole*;
    ⟨ Implementation for `librenv.pas` 467 ⟩
    begin *InitLibrEnv*; *CheckCompatibility*;
  end.

464. ⟨ Interface for `MIZFILES` library 464 ⟩ ≡
const *MML* = ´`mml`´; *EnvMizFiles* = ´`MIZFILES`´;
var *MizPath*, *MizFiles*: *string*;
function *LibraryPath*(*fName*, *fExt* : *string*): *string*;
procedure *GetSortedNames*(*fParam* : *byte*; **var** *fList* : *MStringCollection*);
procedure *GetNames*(*fParam* : *byte*; **var** *fList* : *StringColl*);
procedure *ReadSortedNames*(*fName* : *string*; **var** *fList* : *MStringCollection*);
procedure *ReadNames*(*fName* : *string*; **var** *fList* : *StringColl*);

See also section 465.

This code is used in section 463.

465. There are two public-facing classes.

⟨ Interface for `MIZFILES` library 464 ⟩ +≡
type ⟨ Declare *FileDescr* data type 466 ⟩
  ⟨ Declare *FileDescrCollection* data type 469 ⟩
var *LocFilesCollection*: *FileDescrCollection*;

466. File descriptors. We use file descriptors for things. These are just "a file name" and "a timestamp".

⟨ Declare *FileDescr* data type 466 ⟩ ≡
  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 465.

467. Constructor.

⟨ Implementation for `librenv.pas` 467 ⟩ ≡
constructor *FileDescr.Init*(*fIdent* : *string*; *fTime* : *LongInt*);
  begin *nName* ← *NewStr*(*fIdent*); *Time* ← *fTime*;
  end;

See also sections 470, 479, and 482.

This code is used in section 463.

468. Destructor.

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

469. Collection of file descriptions.

⟨ Declare *FileDescrCollection* data type 469 ⟩ ≡
  PFileDescrCollection = ↑*FileDescrCollection*;
  FileDescrCollection = **object** (*MSortedCollection*)
    function *Compare*(*Key1*, *Key2* : *Pointer*): *integer*; *virtual*;
    procedure *StoreFIL*(*fName* : *string*);
    constructor *LoadFIL*(*fName* : *string*);
    procedure *InsertTimes*;
  end ;

This code is used in section 465.

470. Comparing two entries in a file descriptor collection amounts to comparing the names for the file descriptors.

⟨ Implementation for `librenv.pas` 467 ⟩ +≡
function *FileDescrCollection.Compare*(*Key1*, *Key2* : *Pointer*): *integer*;
  begin if *PFileDescr*(*Key1*)↑.*nName*↑ < *PFileDescr*(*Key2*)↑.*nName*↑ **then** *Compare* ← −1
  else if *PFileDescr*(*Key1*)↑.*nName*↑ = *PFileDescr*(*Key2*)↑.*nName*↑ **then** *Compare* ← 0
    else *Compare* ← 1;
  end;

471. Inserting file times into the file descriptors relies upon `mizenv.pas`'s *GetFileTime* (§37) function.

procedure *FileDescrCollection.InsertTimes*;
  var *z*: *integer*;
  begin for *z* ← 0 **to** *Count* − 1 **do**
    with *PFileDescr*(*Items*↑[*z*])↑ **do** *Time* ← *GetFileTime*(*nName*↑);
  end;

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

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

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

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

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

procedure *FileDescrCollection.StoreFIL*(*fName* : *string*);
  var *FIL*: *text*; *i*: *integer*;
  begin *EraseFile*(*fName*); *Assign*(*FIL*, *fName*); *Rewrite*(*FIL*); *InsertTimes*;
  for $i \leftarrow 0$ **to** $Count - 1$ **do**
    with *PFileDescr*(*Items*↑[*i*])↑ **do**
      begin *WriteLn*(*FIL*, *nName*↑); *WriteLn*(*FIL*, *Time*)
      end;
  Close(*FIL*);
  end;

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

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

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

475. This function actually is not used anywhere, so I am not sure why we have it.

```
procedure ReadSortedNames(fName : string; var fList : MStringCollection);
  var NamesFile: text;
  begin if fName[1] = ´@´ then
    begin Delete(fName, 1, 1); FileExam(fName); Assign(NamesFile, fName); Reset(NamesFile);
    fList.Init(100, 100);
    while ¬seekEof(NamesFile) do
      begin ReadLn(NamesFile, fName); fList.Insert(NewStr(fName));
      end;
    exit;
    end;
  fList.Init(2, 10); fList.Insert(NewStr(fName));
  end;
```

476. Again, this function is not used anywhere, so I am not sure why we have it.

```
procedure ReadNames(fName : string; var fList : StringColl);
  var NamesFile: text;
  begin if fName[1] = ´@´ then
    begin Delete(fName, 1, 1); FileExam(fName); Assign(NamesFile, fName); Reset(NamesFile);
    fList.Init(10, 10);
    while ¬seekEof(NamesFile) do
      begin ReadLn(NamesFile, fName); fList.Insert(NewStr(fName));
      end;
    exit;
    end;
  fList.Init(2, 10); fList.Insert(NewStr(fName));
  end;
```

477. This function is used in `lisvoc.dpr`

```
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 ← ParamStr(fParam);
  if FileName[1] = ´@´ then
    begin Delete(FileName, 1, 1); FileExam(FileName); Assign(NamesFile, FileName);
    Reset(NamesFile); fList.Init(10, 10);
    while ¬seekEof(NamesFile) do
      begin ReadLn(NamesFile, FileName); fList.Insert(NewStr(TrimString(FileName)));
      end;
    exit;
    end;
  fList.Init(2, 8); fList.Insert(NewStr(FileName));
  for i ← fParam + 1 to ParamCount do
    begin FileName ← ParamStr(i); fList.Insert(NewStr(FileName));
    end;
  end;
```

478. Continuing with the "this is not used anywhere" theme, this function is not used anywhere.

procedure *GetNames*(*fParam* : *byte*; **var** *fList* : *StringColl*);
  var *FileName*: *string*; *NamesFile*: *text*; *i*: *integer*;
  begin if *ParamCount* < *fParam* **then**
    begin *fList*.*Init*(0, 0); *exit*
    end;
  FileName ← *ParamStr*(*fParam*);
  if *FileName*[1] = ´@´ **then**
    begin *Delete*(*FileName*, 1, 1); *FileExam*(*FileName*); *Assign*(*NamesFile*, *FileName*);
    Reset(*NamesFile*); *fList*.*Init*(10, 10);
    while ¬*seekEof*(*NamesFile*) **do**
      begin *ReadLn*(*NamesFile*, *FileName*); *fList*.*Insert*(*NewStr*(*TrimString*(*FileName*)));
      end;
    exit;
    end;
  fList.*Init*(2, 8); *fList*.*Insert*(*NewStr*(*FileName*));
  for *i* ← *fParam* + 1 **to** *ParamCount* **do**
    begin *FileName* ← *ParamStr*(*i*); *fList*.*Insert*(*NewStr*(*FileName*));
    end;
  end;

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

The `mml.ini` file looks something like:

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

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

define *init_val_and_end*(#) ≡ *val*(*lLine*, #, *lCode*);
  end
define *Try_read_ini_var*(#) ≡ *lPos* ← *Pos*(#, *lLine*);
  if *lPos* > 0 **then**
   begin *delete*(*lLine*, 1, *lPos* + 15); *init_val_and_end*

⟨ Implementation for `librenv.pas` 467 ⟩ +≡
procedure *CheckCompatibility*;
 var *lFile*: *text*; *lLine*, *lVer1*, *lVer2*, *l*: *string*; *lPos*, *lCode*: *integer*;
  lMizarReleaseNbr, *lMizarVersionNbr*, *lMizarVariantNbr*: *integer*;
 begin ⟨ Open `mml.ini` file 480 ⟩
 lMizarReleaseNbr ← −1; *lMizarVersionNbr* ← −1; *lMizarVariantNbr* ← −1;
 while ¬*seekEof*(*lFile*) **do**
  begin *ReadLn*(*lFile*, *lLine*); *Try_read_ini_var*(´`MizarReleaseNbr=`´)(*lMizarReleaseNbr*);
  Try_read_ini_var(´`MizarVersionNbr=`´)(*lMizarVersionNbr*);
  Try_read_ini_var(´`MizarVariantNbr=`´)(*lMizarVariantNbr*);
  end;
 close(*lFile*);
 ⟨ Assert MML version is compatible with Mizar version 481 ⟩
 end;

480. We open the `$MIZFILES/mml.ini` file for reading.

⟨ Open `mml.ini` file 480 ⟩ ≡
 FileExam(*MizFiles* + *MML* + ´`.ini`´); *Assign*(*lFile*, *MizFiles* + *MML* + ´`.ini`´); *Reset*(*lFile*);

This code is used in section 479.

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

⟨ Assert MML version is compatible with Mizar version 481 ⟩ ≡
 if ¬((*lMizarReleaseNbr* = *PCMizarReleaseNbr*) ∧ (*lMizarVersionNbr* = *PCMizarVersionNbr*)) **then**
  begin *Str*(*PCMizarReleaseNbr*, *l*); *lVer1* ← *l*; *Str*(*PCMizarVersionNbr*, *l*); *lVer1* ← *lVer1* + ´`.`´ + *l*;
  Str(*PCMizarVariantNbr*, *l*); *lVer1* ← *lVer1* + ´`.`´ + *l*; *Str*(*lMizarReleaseNbr*, *l*); *lVer2* ← *l*;
  Str(*lMizarVersionNbr*, *l*); *lVer2* ← *lVer2* + ´`.`´ + *l*;
  Str(*lMizarVariantNbr*, *l*); *lVer2* ← *lVer2* + ´`.`´ + *l*; *DrawMessage*(´`Mizar␣System␣ver.␣`´ + *lVer1* +
   ´`␣is␣incompatible␣with␣the␣MML␣version␣imported␣(`´ + *lVer2* + ´`)`´,
   ´`Please␣check␣`´ + *MizFiles* + ´`mml.ini`´); *halt*(1);
  end;

This code is used in section 479.

482. 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*(#) ≡ **if** #[*length*(#)] ≠ *DirSeparator* **then** # ← # + *DirSeparator*;

⟨ Implementation for `librenv.pas` 467 ⟩ +≡
procedure *InitLibrEnv*;
  begin *LocFilesCollection*.*Init*(0, 20); *MizPath* ← *ExtractFileDir*(*ParamStr*(0)); ⟨ Initialize *Mizfiles* 483 ⟩
  MizFileName ← ´´; *EnvFileName* ← ´´; *ArticleName* ← ´´; *ArticleExt* ← ´´;
  end;

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

⟨ Initialize *Mizfiles* 483 ⟩ ≡
  ⟨ Guess *MizFiles* from environment variables or executable path 484 ⟩
  if *MizFiles* ≠ ´´ **then** *append_dir_separator*(*MizFiles*);
  if *MizFiles* = ´´ **then** *Mizfiles* ← *DirSeparator*;

This code is used in section 482.

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

⟨ Guess *MizFiles* from environment variables or executable path 484 ⟩ ≡
  MizFiles ← *GetEnvStr*(*EnvMizFiles*);
  if *MizFiles* = ´´ **then** *MizFiles* ← *MizPath*;

This code is used in section 483.

File 12

Info file handling

485. I don't think this is actually used anywhere, but I am including it for completeness.

```
⟨ info.pas 485 ⟩ ≡
  ⟨ 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);
  procedure 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, ´␣´)
    end;
  procedure InfoWord(C : char; I : integer);
    begin write(InfoFile, C, I, ´␣´)
    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;
```

486. There are a few helper functions which is more than "Write ⟨data type⟩ to info file".

var *_InfoExitProc*: *pointer*;
procedure *InfoExitProc*;
  begin *CloseInfoFile*; *ExitProc* ← *_InfoExitProc*;
  end;
procedure *OpenInfoFile*;
  begin *Assign*(*InfoFile*, *MizFileName* + ´`.inf`´); *Rewrite*(*InfoFile*);
  WriteLn(*InfoFile*, ´`Mizared␣article:␣"`´, *MizFileName*, ´`"`´); *_InfoExitProc* ← *ExitProc*;
  ExitProc ← @*InfoExitProc*;
  end;
procedure *CloseInfofile*;
  begin *close*(*InfoFile*)
  end;

  end .

File 13

XML Parser

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

Just to review some terminology from XML:

(1) A **"tag"** is a markup construct that begins with a "`<`" and ends with a "`>`". There are three types of tags:
  (i) Start-tags: like "`<foo>`"
  (ii) End-tags: like "`</foo>`"
  (iii) Empty-element tags: like "`<br />`"

(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 "`<img src="madonna.jpg" alt="Madonna" />`" has two attributes: one named "src" whose value is "madonna.jpg", and the other named "alt" whose value is "Madonna".

(4) XML documents may start with an **"XML declaration"** which looks something like (after some optional whitespace) "`<?xml version="1.0" encoding="UTF-8"?>`"

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

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

⟨ Constants for `xml_parser.pas` 488 ⟩ ≡
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 487.

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

⟨ Type declarations for `xml_parser.pas` 489 ⟩ ≡
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 = **set of** *XMLTokenKind*;
  ⟨ Declare XML Scanner Object type 494 ⟩
  TElementState = (*eStart*, *eEnd*); { high-level parser states, see procedure NextElementState }
  ⟨ Declare XML Attribute Object 490 ⟩
  ⟨ Declare XML Parser object 502 ⟩

This code is used in section 487.

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

⟨ Declare XML Attribute Object 490 ⟩ ≡
  XMLAttrPtr = ↑*XMLAttrObj*;
  XMLAttrObj = **object** (*MStrObj*)
    nValue: *string*;
    constructor *Init*(**const** *aName*, *aValue*: *string*);
  end ;

This code is used in section 489.

491. Constructor. This uses the *MStrObj.Init* constructor to initialize the name, then it sets the value.

⟨ Implementation of XML Parser 491 ⟩ ≡

```
constructor XMLAttrObj.Init(const aName, aValue: string);
  begin inherited Init(aName); nValue ← aValue;
  end;
```

See also sections 492, 493, 495, 497, 498, 501, 503, 508, and 510.

This code is used in section 487.

492. Assertion. We have a helper function for asserting things about XML. This is just a wrapper around *MizAssert* (§113).

⟨ Implementation of XML Parser 491 ⟩ +≡

```
procedure XMLASSERT(aCond : boolean);
  begin MizAssert(errWrongXMLElement, aCond);
  end;
```

493. Unexpected XML Element. Another helper function for checking XML parsing.

⟨ Implementation of XML Parser 491 ⟩ +≡

```
procedure UnexpectedXMLElem(const aElem: string;
  aErr: integer);
    mdebug ;
  var lEl: string;
    end_mdebug ;
  begin
  mdebug ; InfoNewLine; end_mdebug;
  RunTimeError(aErr);
  end;
```

494. XML Scanner Object. The scanner produces a stream of tokens, which is then consumed by the XML parser. Hence, besides the constructor and destructor, there is only one public facing method: get the next token.

⟨ Declare XML Scanner Object type 494 ⟩ ≡

```
  XMLScannObj = object (MObject)
    nSourceFile: text;
    nSourceFileBuff: pointer;
    nCurTokenKind: XMLTokenKind;
    nSpelling: string;
    nPos: Position;
    nCurCol: integer;
    nLine: string;
    constructor InitScanning(const aFileName: string);
    destructor Done; virtual;
    procedure GetToken; private
    procedure GetAttrValue;
  end ;
```

This code is used in section 489.

495. 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.

⟨Implementation of XML Parser 491⟩ +≡
constructor *XMLScannObj.InitScanning*(**const** *aFileName*: *string*);
  begin *inherited Init*; ⟨Prepare to read in the contents of XML file 496⟩;
  nSpelling ← ´´; *nLine* ← ´´; *nCurCol* ← 0; *nPos.Line* ← 0; *nPos.Col* ← 0;
  GetToken;
  end;

496. This prepares to read in from an XML file, setting up a text buffer, and opening the file in "read mode".

⟨Prepare to read in the contents of XML file 496⟩ ≡
  Assign(*nSourceFile*, *aFileName*); *GetMem*(*nSourceFileBuff*, *InOutFileBuffSize*);
  SetTextBuf(*nSourceFile*, *nSourceFileBuff*↑, *InOutFileBuffSize*); *Reset*(*nSourceFile*) { open for reading }

This code is used in section 495.

497. Destructor. We need to close the XML file, as well as free up the input buffer.

⟨Implementation of XML Parser 491⟩ +≡
destructor *XMLScannObj.Done*;
  begin *close*(*nSourceFile*); *FreeMem*(*nSourceFileBuff*, *InOutFileBuffSize*);
  nLine ← ´´; *nSpelling* ← ´´;
  inherited Done;
  end;

498. 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.

  define *update_lexeme* ≡ *nSpelling* ← *Copy*(*nLine*, *nPos.Col*, *nCurCol* − *nPos.Col*)

⟨Implementation of XML Parser 491⟩ +≡
procedure *XMLScannObj.GetToken*;
  const *CharKind*: **array** [*chr*(0) .. *chr*(255)] **of**
      byte = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    {   #   &   - . / 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,
    { 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   _ }
      0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 3,
    { 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 }
      0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
  begin ⟨Skip whitespace for XML parser 499⟩;
  nPos.Col ← *nCurCol*;
  ⟨Get token kind based off of leading character 500⟩;
  update_lexeme;
  while (*nCurCol* < *length*(*nLine*)) ∧ (*nLine*[*nCurCol*] ∈ [´␣´, ´ ´]) **do** *inc*(*nCurCol*);
  end;

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

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

⟨ Skip whitespace for XML parser 499 ⟩ ≡

```
while nCurCol = length(nLine) do
  begin if eof(nSourceFile) then
    begin nCurTokenKind ← EOTX; nSpelling ← ´´; exit end;
  ReadLn(nSourceFile, nLine); inc(nPos.Line); nLine ← nLine + ´␣´; nCurCol ← 1;
  while (nCurCol < length(nLine)) ∧ (nLine[nCurCol] = ´␣´) do inc(nCurCol);
  end
```

This code is used in section 498.

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

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

```
define keep_eating_alphadigits ≡
        begin nCurTokenKind ← ID;
        repeat inc(nCurCol)
        until CharKind[nLine[nCurCol]] = 0;
        end
define get_tag_kind ≡ inc(nCurCol);
      case nLine[nCurCol] of
      ´/´: begin nCurTokenKind ← ET; inc(nCurCol); end;
      ´?´: begin nCurTokenKind ← BI; inc(nCurCol); end;
      ´!´: begin nCurTokenKind ← DT; inc(nCurCol); end;
      othercases nCurTokenKind ← LT;
      endcases
define keep_getting_until_end_of_tag(#) ≡ begin inc(nCurCol);
      if nLine[nCurCol] = ´>´ then
        begin nCurTokenKind ← #; inc(nCurCol); end
      else nCurTokenKind ← Err;
      end;
```

⟨ Get token kind based off of leading character 500 ⟩ ≡

```
case nLine[nCurCol] of
´a´ .. ´z´, ´A´ .. ´Z´, ´0´ .. ´9´, ´_´, ´-´, ´&´: keep_eating_alphadigits;
´"´: begin nCurTokenKind ← QT; inc(nCurCol) end;
´=´: begin nCurTokenKind ← EQ; inc(nCurCol) end;
´<´: begin get_tag_kind; end;
´>´: begin nCurTokenKind ← GT; inc(nCurCol) end;
´/´: keep_getting_until_end_of_tag(EE);
´?´: keep_getting_until_end_of_tag(EI);
othercases begin nCurTokenKind ← Err; inc(nCurCol) end;
endcases
```

This code is used in section 498.

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

This will not include the delimiting quotation marks, and it will also skip all whitespace *after* the attribute.

define *skip_to_quotes* ≡ **while** (*nCurCol* < *length*(*nLine*)) ∧ (*nLine*[*nCurCol*] ≠ ´"´) **do** *inc*(*nCurCol*)
define *is_space* ≡ (*nCurCol* < *length*(*nLine*)) ∧ (*nLine*[*nCurCol*] ∈ [´␣´, ´ ´])
define *skip_spaces* ≡ **while** *is_space* **do** *inc*(*nCurCol*)

⟨Implementation of XML Parser 491⟩ +≡
procedure *XMLScannObj.GetAttrValue*;
  var *lCol*: *integer*;
  begin *lCol* ← *nCurCol*; *skip_to_quotes*;
  nSpelling ← *Copy*(*nLine*, *lCol*, *nCurCol* − *lCol*); { save the lexeme }
  if *nLine*[*nCurCol*] = ´"´ **then** *inc*(*nCurCol*);
  skip_spaces;
  end;

502. XML Parser. We recall (§489) the type for element states (it's an enumerated type with two values, *eStart* and *eEnd*).

⟨Declare XML Parser object 502⟩ ≡
  XMLParserObj = **object** (*XMLScannObj*)
    nElName: *string*; { name of the current element }
    nState: *TElementState*;
    nAttrVals: *MSortedStrList*;
    constructor *InitParsing*(**const** *aFileName*: *string*);
    destructor *Done*; *virtual*;
    procedure *ErrorRecovery*(*aErr* : *integer*; *aSym* : *TokensSet*);
    procedure *NextTag*; *virtual*;
    procedure *NextElementState*; *virtual*;
    procedure *AcceptEndState*; *virtual*;
    procedure *AcceptStartState*; *virtual*;
    procedure *OpenStartTag*; *virtual*;
    procedure *CloseStartTag*; *virtual*;
    procedure *CloseEmptyElementTag*; *virtual*;
    procedure *ProcessEndTag*; *virtual*;
    procedure *ProcessAttributeName*; *virtual*;
    procedure *ProcessAttributeValue*; *virtual*;
    procedure *SetAttributeValue*(**const** *aVal*: *string*);
  end ;

This code is used in section 489.

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

The constructor will then skip all other "`<?...?>`" entities.

```
define skip_xml_prolog ≡ while (nCurTokenKind ≠ EOTX) ∧ (nCurTokenKind ≠ EI) do GetToken;
        if nCurTokenKind = EI then GetToken
define skip_all_other_ids ≡ while nCurTokenKind = BI do
          begin GetToken;
          while (nCurTokenKind ≠ EOTX) ∧ (nCurTokenKind ≠ EI) do GetToken;
          if nCurTokenKind = EI then GetToken;
          end
```

⟨Implementation of XML Parser 491⟩ +≡

```
constructor XMLParserObj.InitParsing(const aFileName: string);
  begin inherited InitScanning(aFileName); nElName ← ´´; nAttrVals.Init(0);
  if nCurTokenKind = BI then
    begin GetToken;
    if (nCurTokenKind = ID) ∧ (nSpelling = ´xml´) then GetToken
    else ErrorRecovery(10, [EI, LT]);
    skip_xml_prolog; skip_all_other_ids;   { skip all other initial processing instructions }
    end;
  end;
```

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

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

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

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

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

```
define report_mismatch(#) ≡ ErrorRecovery(#)
define xml_match(#) ≡ if nCurTokenKind ≠ # 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;
```

506. The parser will the consume the next tag or element. It's useful to recall the token kinds (§489).
Curiously, the attributes are skipped during this parsing function.
This will be using the inherited procedure *GetToken* (§498).

{ Parses next part of XML, used for skipping some part of XML }
{ setting the *nState* to *eStart* or *eEnd*. }
{ *nElName* is set properly }
{ *nAttrVals* are omitted (skiped). }

procedure *XMLParserObj.NextTag*;
begin case *nCurTokenKind* **of**
EOTX: *nState* ← *eEnd*; { sometimes we need this }
LT: **begin** *nState* ← *eStart*; *GetToken*; *xml_match*(*ID*)(6, [*LT*, *ET*]); *OpenStartTag*; *GetToken*;
⟨ Get contents of XML start tag 507 ⟩;
end;
EE: **begin** *nState* ← *eEnd*; *GetToken*; **end**;
ET: **begin** *nState* ← *eEnd*; *GetToken*; *xml_match*(*ID*)(8, [*LT*, *ET*]); *OpenStartTag*; *GetToken*;
xml_match(*GT*)(7, [*LT*, *ET*]); *GetToken*
end;
othercases *ErrorRecovery*(9, [*LT*, *ET*]);
endcases;
end;

507. When getting the contents of an XML start tag (or possibly an element), we keep going until we get to either "\>" (for an element) or ">" (for a tag). This will be using the inherited procedure *GetToken* (§498).

define *get_attribute* ≡ **begin** *GetToken*; *xml_match*(*EQ*)(4, [*ID*, *GT*, *LT*, *ET*]); *GetToken*;
xml_match(*QT*)(3, [*ID*, *GT*, *LT*, *ET*]); *GetAttrValue*; *GetToken*;
end

⟨ Get contents of XML start tag 507 ⟩ ≡
repeat case *nCurTokenKind* **of**
GT: **begin** *GetToken*; *break* **end**;
EE: **begin** *break* **end**;
ID: *get_attribute*;
othercases begin *ErrorRecovery*(5, [*GT*, *LT*, *ET*]); *break* **end**;
endcases;
until *nCurTokenKind* = *EOTX*

This code is used in section 506.

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

⟨ Implementation of XML Parser 491 ⟩ +≡

```
   { Parses next part of XML, setting the nState to eStart or eEnd. If nState = eStart, then nElName,
     nAttrVals are set properly. It is possible to go from nState = eStart to nState = eStart (when the
     element is non empty), and similarily from eEnd to eEnd. }
procedure XMLParserObj.NextElementState;
  begin case nCurTokenKind of
  EOTX: nState ← eEnd;  { sometimes we need this }
  LT: ⟨ Parse start of XML tag 509 ⟩;
  EE: begin nState ← eEnd; GetToken; end;
  ET: begin nState ← eEnd; GetToken; xml_match(ID)(8, [LT, ET]); ProcessEndTag; GetToken;
    xml_match(GT)(7, [LT, ET]); GetToken; end;
  othercases ErrorRecovery(9, [LT, ET]);
  endcases;
  end;
```

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

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

```
  define end_start_tag ≡ begin GetToken; CloseStartTag; break end
  define end_empty_tag ≡ begin CloseEmptyElementTag; break end
```

⟨ Parse start of XML tag 509 ⟩ ≡

```
  begin nState ← eStart; GetToken; xml_match(ID)(6, [LT, ET]); OpenStartTag;
      { 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: begin ProcessAttributeName; GetToken; xml_match(EQ)(4, [ID, GT, LT, ET]); GetToken;
      xml_match(QT)(3, [ID, GT, LT, ET]); GetAttrValue; ProcessAttributeValue; GetToken;
      end;
    othercases begin ErrorRecovery(5, [GT, LT, ET]); break end;
    endcases;
  until nCurTokenKind = EOTX;
  end
```

This code is used in section 508.

510. We will want assertions reflecting the parser is in a "start" state or an "end" state.

⟨ Implementation of XML Parser 491 ⟩ +≡

```
procedure XMLParserObj.AcceptEndState;
  begin NextElementState; MizAssert(errElRedundant, nState = eEnd);
  end;

procedure XMLParserObj.AcceptStartState;
  begin NextElementState; MizAssert(errElMissing, nState = eStart);
  end;
```

511.

procedure *XMLParserObj.OpenStartTag*;
  begin *nElName* ← *nSpelling*; *nAttrVals.FreeAll*;
  end;

512. We have a few procedures which are, well, empty. I am not sure why we have them. Regardless, here they are!

procedure *XMLParserObj.CloseStartTag*;
  begin end;

procedure *XMLParserObj.CloseEmptyElementTag*;
  begin end;

procedure *XMLParserObj.ProcessEndTag*;
  begin end;

513. We have a list of attributes. When the parser *ProcessAttributeName*, it will merely push a new *XMLAttrPtr* to the list with the given name. Then *ProcessAttributeValue* will associate to it the value which has been parsed. We can, of course, *manually* set the value for an attribute using *SetAttributeValue*.

procedure *XMLParserObj.ProcessAttributeName*;
  begin *nAttrVals.Insert*(*new*(*XMLAttrPtr*, *Init*(*nSpelling*, ´´)));
  end;

procedure *XMLParserObj.ProcessAttributeValue*;
  begin *SetAttributeValue*(*nSpelling*);
  end;

procedure *XMLParserObj.SetAttributeValue*(**const** *aVal*: *string*);
  begin with *nAttrVals* **do** *XMLAttrPtr*(*Items*↑[*Count* − 1])↑.*nValue* ← *aVal*;
  end;

File 14

I/O with XML

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

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

⟨ xml_inout.pas 514 ⟩ ≡
 ⟨ GNU License 4 ⟩
unit *xml_inout*;
 interface
 uses *errhan*, *mobjects*, *xml_parser*;
 ⟨ Type declarations for XML I/O 515 ⟩
 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 516 ⟩
end .

515. There are only 4 types of streams we care about: Streams, Text Streams, XML Input Streams, and XML Output Streams.

⟨ Type declarations for XML I/O 515 ⟩ ≡
 ⟨ Public interface for XML Input Stream 528 ⟩;
 ⟨ Public declaration for Stream Object 520 ⟩;
 ⟨ Public declaration for Text Stream Object 524 ⟩;
 ⟨ Public declaration for XML Output Stream 533 ⟩;

This code is used in section 514.

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

```
⟨Implementation for I/O of XML 516⟩ ≡
function QuoteStrForXML(const aStr: string): string;
  const ValidCharTable = (['a' .. 'z', 'A' .. 'Z', '0' .. '9', '-', '␣', ',', '.', '\', '/', '_', '[', ']',
        '!', ';', ':', '=']);
  var c: char; i: integer;
  begin Result ← aStr;
  for i ← length(Result) downto 1 do
    begin c ← Result[i];
    if ¬(c ∈ ValidCharTable) then
      begin Result[i] ← '&'; Insert('#x' + IntToHex(Ord(c), 2) + ';', Result, i + 1);
      end;
    end;
  end;
```

See also sections 519, 521, 525, 529, 534, and 540.

This code is used in section 514.

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

```
function XMLToStr(const aXMLStr: string): string;
  var i, h: integer; lHexNr: string;
  begin Result ← aXMLStr;
  for i ← length(Result) − 5 downto 1 do
    begin ⟨Transform XML entity into character, if encountering an XML entity at i 518⟩;
    end;
  Result ← Trim(Result);
  end;
```

518. Transforming an XML entity into a character. This specifically checks for *hexadecimal* entities of the form “&#xXX” for some hexadecimal digits X. Note we must prepend “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 518⟩ ≡
  if (Result[i] = '&') ∧ (length(Result) ≥ i + 5) then
    begin if (Result[i + 1] = '#') ∧ (Result[i + 2] = 'x') then
      begin lHexNr ← Result[i + 3] + Result[i + 4]; h ← StrToInt('0x' + lHexNr); Delete(Result, i, 5);
      Result[i] ← chr(h);
      end;
    end
```

This code is used in section 517.

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

```
⟨ Implementation for I/O of XML 516 ⟩ +≡
function QuoteXMLAttr(aStr : string): string;
  var i: integer;
  begin Result ← ´´;
  for i ← 1 to length(aStr) do
    case aStr[i] of
    ´"´: Result ← Result + ´&quot;´;
    ´&´: Result ← Result + ´&amp;´;
    ´<´: Result ← Result + ´&lt;´;
    ´>´: Result ← Result + ´&gt;´;
    othercases if integer(aStr[i]) > 127 then
        Result ← Result + ´&#x´ + IntToHex(Ord(aStr[i]), 2) + ´;´
      else Result ← Result + aStr[i];
    endcases;
  end;
```

520. 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.

```
⟨ Public declaration for Stream Object 520 ⟩ ≡
  StreamObj = object (MObject)
    nFile: File;
    fFileBuff: ↑BuffChar;
    fBuffCount, fBuffInd: longint;
    constructor InitFile(const AFileName: string);
    procedure Error(Code, Info : integer); virtual;
    destructor Done; virtual;
  end
```

This code is used in section 515.

521. We will have a wrapper function for conveniently reporting errors.

```
⟨ Implementation for I/O of XML 516 ⟩ +≡
procedure StreamObj.Error(Code, Info : integer);
  begin RunError(2000 + Code);
  end;
```

522. Constructor. We begin by *Assign*-ing a name to a file, allocating a file buffer, then initializing the 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 ← 0; fBuffInd ← 0;
  end;
```

523. Destructor. We close the file, and free up the file buffer.

```
destructor StreamObj.Done;
  begin Close(nFile); dispose(fFileBuff);
  end;
```

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

⟨ Public declaration for Text Stream Object 524 ⟩ ≡
  TXTStreamObj = **object** (*MObject*)
    nFile: *text*;
    nFileBuff: *pointer*;
    constructor *InitFile*(**const** *AFileName*: *string*);
    procedure *Error*(*Code*, *Info* : *integer*); *virtual*;
    destructor *Done*; *virtual*;
  end

This code is used in section 515.

525. We have the convenience function for reporting errors.

⟨ Implementation for I/O of XML 516 ⟩ +≡
procedure *TXTStreamObj.Error*(*Code*, *Info* : *integer*);
  begin *RunError*(2000 + *Code*);
  end;

526. Constructor. Assign a name to the file, allocate an input buffer, then initialize the buffer.

constructor *TXTStreamObj.InitFile*(**const** *AFileName*: *string*);
  begin *Assign*(*nFile*, *AFileName*); *GetMem*(*nFileBuff*, *InOutFileBuffSize*);
  SetTextBuf(*nFile*, *nFileBuff*↑, *InOutFileBuffSize*);
  end;

527. Destructor. Simply free the underlying file buffer.

destructor *TXTStreamObj.Done*;
  begin *FreeMem*(*nFileBuff*, *InOutFileBuffSize*);
  end;

528. XML Input Streams. An input stream reads an XML file and produces an abstract syntax tree for its contents. This extends this XML parser class (§502). It may be tempting to draw similarities with, e.g., the StAX library (in Java), but the truth is there's only finitely many ways to parse XML, and some ways are just more natural.

⟨ Public interface for XML Input Stream 528 ⟩ ≡
  XMLInStreamPtr = ↑*XMLInStreamObj*;
  XMLInStreamObj = **object** (*XMLParserObj*)
    constructor *OpenFile*(**const** *AFileName*: *string*);
    function *GetOptAttr* (**const** *aAttrName*: *string*; **var** *aVal*: *string*) : *boolean*;
    function *GetAttr*(**const** *aAttrName*: *string*): *string*;
    function *GetIntAttr*(**const** *aAttrName*: *string*): *integer*;
  end

This code is used in section 515.

529. Constructor. The non-debugging code just invokes the XML Parser's constructor (§503).

⟨ Implementation for I/O of XML 516 ⟩ +≡
constructor *XMLInStreamObj.OpenFile*(**const** *AFileName*: *string*);
  begin
  mdebug ; *write*(*InfoFile*, *AFileName*); **end_mdebug**;
  InitParsing(*AFileName*);
  mdebug ; *WriteLn*(*InfoFile*, ´␣reset´); **end_mdebug**;
  end;

530. We use the inherited *XMLParserObj*'s *nAttrVals*: *MSortedStrList* to track the XML attributes. If *aAttrName* is stored there, this will mutate *aVal* to store the associated value and the function will return *true*. Otherwise, this will return *false*.

This is useful for getting the value of an *optional* XML attribute.

```
{ get string denoted by optional XML attribute aAttrName }
function XMLInStreamObj.GetOptAttr (const aAttrName: string; var aVal: string) : boolean;
  var lAtt: XMLAttrPtr;
    begin lAtt ← XMLAttrPtr(nAttrVals.ObjectOf(aAttrName));
    if lAtt ≠ nil then
      begin aVal ← lAtt↑.nValue; GetOptAttr ← true; exit;
      end;
    GetOptAttr ← false;
    end;
```

531. When we know an XML attribute is *required*, we can just get the associated value directly (and raise an error if it is missing).

```
{ get string denoted by required XML attribute aAttrName }
function XMLInStreamObj.GetAttr(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;
```

532. When the required attribute has an integer value, we should return the integer-value of it. Does this ever happen? Yes! For example, when writing an article named `article.miz`, then we run the verifier on it, we shall obtain `article.xml` which will contain tags of the form "`<Adjective nr="5">`".

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

533. XML Output Streams. We will want to write data to an XML file. This gives us an abstraction for doing so.

⟨ Public declaration for XML Output Stream 533 ⟩ ≡
```
XMLOutStreamPtr = ↑XMLOutStreamObj;
XMLOutStreamObj = object (StreamObj)
  nIndent: integer; { indenting }
  constructor OpenFile(const AFileName: string);
  constructor OpenFileWithXSL(const AFileName: string);
  destructor EraseFile;
  procedure OutChar(AChar : char);
  procedure OutNewLine;
  procedure OutString(const AString: string);
  procedure OutIndent;
  procedure Out_XElStart(const fEl: string);
  procedure Out_XAttrEnd;
  procedure Out_XElStart0(const fEl: string);
  procedure Out_XElEnd0;
  procedure Out_XEl1(const fEl: string);
  procedure Out_XElEnd(const fEl: string);
  procedure Out_XAttr(const fAt, fVal: string);
  procedure Out_XIntAttr(const fAt: string;
    fVal: integer);
  procedure Out_PosAsAttrs(const fPos: Position);
  procedure Out_XElWithPos(const fEl: string;
    const fPos: Position);
  procedure Out_XQuotedAttr(const fAt, fVal: string);
  destructor Done; virtual;
end
```
This code is used in section 515.

534. Constructor. We initialize a file, open it for writing, set the initial indentation amount to zero, and then print the XML header declaration.

⟨ Implementation for I/O of XML 516 ⟩ +≡
```
constructor XMLOutStreamObj.OpenFile(const AFileName: string);
begin
  mdebug write(InfoFile, MizFileName + ´.´ + copy(AFileName, length(AFilename) − 2, 3));
    end_mdebug
  InitFile(AFileName); Rewrite(nFile, 1);
  mdebug WriteLn(InfoFile, ´␣rewritten´); end_mdebug
  nIndent ← 0; OutString(gXMLHeader);
  end;
```

535. 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 procesing info }
constructor XMLOutStreamObj.OpenFileWithXSL(const AFileName: string);
  begin OpenFile(AFileName);
  OutString(´<?xml-stylesheet␣type="text/xml"␣href="file://´ + MizFiles + ´miz.xml"?>´ + #10);
  end;
```

536. Destructor. We need to flush the buffer to the file before freeing up the buffer.

destructor *XMLOutStreamObj.Done*;
  begin if $(fBuffInd > 0) \wedge (fBuffInd < InOutFileBuffSize)$ **then**
    BlockWrite(*nFile*, *fFileBuff*↑, *fBuffInd*, *fBuffCount*);
  inherited Done;
  end;

537. Destructor. Some times we want to further erase the output file (which seems, at first glance, like *a really bad idea...*).

destructor *XMLOutStreamObj.EraseFile*;
  begin *Done*; *Erase*(*nFile*);
  end;

538. Writing a character to the buffer. When the buffer is full, we flush it.

procedure *XMLOutStreamObj.OutChar*(*aChar* : *char*);
  begin *fFileBuff*↑[*fBuffInd*] ← *AnsiChar*(*aChar*); *inc*(*fBuffInd*); ⟨Flush XML output buffer, if full 539⟩;
  end;

539. 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.

⟨Flush XML output buffer, if full 539⟩ ≡
  if *fBuffInd* = *InOutFileBuffSize* **then**
    begin *BlockWrite*(*nFile*, *fFileBuff*↑, *InOutFileBuffSize*, *fBuffCount*); *fBuffInd* ← 0;
    end

This code is used in section 538.

540. Print a newline (`"\n"`) to the XML output stream.

⟨Implementation for I/O of XML 516⟩ +≡
procedure *XMLOutStreamObj.OutNewLine*;
  begin *OutChar*(#10);
  end;

541. Printing a string to the output buffer.

procedure *XMLOutStreamObj.OutString*(**const** *aString*: *string*);
  var *i*: *integer*;
  begin for *i* ← 1 **to** *length*(*aString*) **do** *OutChar*(*aString*[*i*]);
  end;

542. Printing *nIndent* spaces ("␣") to the output buffer.

{ print *nIndent* spaces }
procedure *XMLOutStreamObj.OutIndent*;
  var *i*: *integer*;
  begin for *i* ← 1 **to** *nIndent* **do** *OutChar*('␣');
  end;

543. When printing a start-tag to the file, we start by printing the indentation, then we increment the indentation, then we print the "<" followed by the name of the tag.

```
{ print '<' and the representation of fEl with indenting }
procedure XMLOutStreamObj.Out_XElStart(const fEl: string);
  begin OutIndent; inc(nIndent); OutChar('<'); OutString(fEl);
  end;
```

544. When we are done writing the attributes of a tag, we print the ">" to the file, and we also print a newline to the file.

```
{ close the attributes with '>' }
procedure XMLOutStreamObj.Out_XAttrEnd;
  begin OutChar('>'); OutNewLine;
  end;
```

545. When we want to write the tag, but omit the attributes, we can do so.

```
{ no attributes expected }
procedure XMLOutStreamObj.Out_XElStart0(const fEl: string);
  begin Out_XElStart(fEl); Out_XAttrEnd;
  end;
```

546. For empty-element tags, we should close the tag with "/>", print a new line, then *decrement* the indentation since there are no children to the tag.

```
{ print '/>' with indenting }
procedure XMLOutStreamObj.Out_XElEnd0;
  begin OutString('/>'); OutNewLine; dec(nIndent);
  end;
```

547. When printing an empty-element tag without any attributes, we can combine the preceding functions together.

```
{ no attributes and elements expected }
procedure XMLOutStreamObj.Out_XEl1(const fEl: string);
  begin Out_XElStart(fEl); Out_XElEnd0;
  end;
```

548. Printing end-tags should first decrement the indentation *before* printing the indentation to the file (so that the end-tag vertically aligns with the associated start-tag). Then we print "</" followed by the tag name and then ">". We should print a newline to the file, too.

```
{ close the fEl element using '</' }
procedure XMLOutStreamObj.Out_XElEnd(const fEl: string);
  begin dec(nIndent); OutIndent; OutString('</'); OutString(fEl); OutChar('>'); OutNewLine;
  end;
```

549. When printing one attribute to a tag, we need a whitespace printed (to separate the tag's name — or preceding attribute — from the current attribute being printed), followed by the attribute's name printed with an equality symbol, then enquoted the value of the attribute.

```
{ print one attribute key-value pair }
procedure XMLOutStreamObj.Out_XAttr(const fAt, fVal: string);
  begin OutChar('␣'); OutString(fAt); OutString('="'); OutString(fVal); OutChar('"');
  end;
```

550. When the value of an attribute is an integer, invoke *IntToStr*(*fVal*) to pretend it is a string value. Then printing out to a file an attribute with an integer value boils down to printing out the attribute with a string value.

{ print one attribute key-value pair, where value is integer }
procedure *XMLOutStreamObj.Out_XIntAttr*(**const** *fAt*: *string*; *fVal*: *integer*);
  begin *Out_XAttr*(*fAt*, *IntToStr*(*fVal*));
  end;

551. We can now just compose writing the start of a tag (§543), followed by its attributes (§552), and then close the empty-element tag (§546).

procedure *XMLOutStreamObj.Out_XElWithPos*(**const** *fEl*: *string*; **const** *fPos*: *Position*);
  begin *Out_XElStart*(*fEl*); *Out_PosAsAttrs*(*fPos*); *Out_XElEnd0*;
  end;

552. We will want to treat a *position* (i.e., the line and column) as two attributes. We print this out using *Out_PosAsAttrs*. We rely on the *XMLDict*'s *XMLAttrName* for standardizing the name for the line and column.

procedure *XMLOutStreamObj.Out_PosAsAttrs*(**const** *fPos*: *Position*);
  begin *Out_XIntAttr*(*XMLAttrName*[*atLine*], *fPos.Line*);
  Out_XIntAttr(*XMLAttrName*[*atCol*], *fPos.Col*);
  end;

553. We print a quoted attribute, leveraging printing attributes out to the file (§549). We just need to escape the XML string (§516).

procedure *XMLOutStreamObj.Out_XQuotedAttr*(**const** *fAt*, *fVal*: *string*);
  begin *Out_XAttr*(*fAt*, *QuoteStrForXML*(*fVal*));
  end;

File 15

Vocabulary file dictionaries

554. Mizar works with vocabulary files (suffixed with `.voc`) for introducing new identifiers.

⟨ dicthan.pas 554 ⟩ ≡
  ⟨ GNU License 4 ⟩
unit *dicthan*;
  interface
  uses *mobjects*;
  ⟨ Public constants for `dicthan.pas` 555 ⟩
  type *SymbolCounters* = **array** [´A´ .. ´Z´] **of** *word*;
    SymbolIntSeqArr = **array** [´A´ .. ´Z´] **of** *IntSequence*;
    ⟨ Class declarations for `dicthan.pas` 556 ⟩
    ⟨ Public function declarations for `dicthan.pas` 557 ⟩
  implementation
  uses *mizenv*, *xml_inout*, *xml_dict*;
  ⟨ Implementation for `dicthan.pas` 558 ⟩
  end .

555. We recall from Adam Grabowski, Artur Korniłowicz, and Adam Naumowicz's "Mizar in a Nutshell" (§4.3, `doi:10.6092/issn.1972-5787/1980`), the various prefixes for vocabulary file entries:
– G for structures
– K for left-functor brackets
– L for right-functor brackets
– M for modes
– O for functors
– R for predicates
– U for selectors
– V for attributes

⟨ Public constants for `dicthan.pas` 555 ⟩ ≡
const
  StandardPriority = 64;
  AvailableSymbols = [´G´, ´K´, ´L´, ´M´, ´O´, ´R´, ´U´, ´V´];

This code is used in section 554.

556. 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.

⟨ Class declarations for `dicthan.pas` 556 ⟩ ≡
  ⟨ Symbol for vocabulary 562 ⟩;
  ⟨ Abstract vocabulary object declaration 571 ⟩;
  ⟨ Vocabulary object declaration 573 ⟩;

This code is used in section 554.

557. ⟨ Public function declarations for `dicthan.pas` 557 ⟩ ≡
function *GetPrivateVoc*(**const** *fName*: *string*): *PVocabulary*;
function *GetPublicVoc* (**const** *fName*: *string*; **var** *fVocFile*: *text*) : *PVocabulary*;
procedure *LoadMmlVcb* (**const** *aFileName*: *string*; **var** *aMmlVcb*: *MStringList*) ;
procedure *StoreMmlVcb*(**const** *aFileName*: *string*; **const** *aMmlVcb*: *MStringList*);
procedure *StoreMmlVcbX*(**const** *aFileName*: *string*; **const** *aMmlVcb*: *MStringList*);

This code is used in section 554.

558. 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* ≡ *Delete*(*lLine*, 1, 1)

⟨ Implementation for `dicthan.pas` 558 ⟩ ≡
function *IsValidSymbol*(**const** *aLine*: *string*): *boolean*;
  var *lLine*: *string*; *lKind*: *char*; *lPriority*, *lPos*, *lCode*: *integer*;
  begin *IsValidSymbol* ← *false*; *lLine* ← *TrimString*(*aLine*);
  ⟨ Initialize *lKind*, but exit if dictionary line contains invalid symbol 559 ⟩;
  delete_prefix;
  case *lKind* **of**
  ´O´: ⟨ Check if functor symbol is valid 560 ⟩;
  ´R´: ⟨ Check if predicate symbol is valid 561 ⟩;
  othercases begin if *Pos*(´␣´, *lLine*) > 0 **then** *exit*;
    IsValidSymbol ← *true*;
    end;
  endcases;
  end;

See also sections 563, 567, 572, 574, 578, 580, and 581.

This code is used in section 554.

559. 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.

⟨ Initialize *lKind*, but exit if dictionary line contains invalid symbol 559 ⟩ ≡
  if *length*(*lLine*) ≤ 1 **then** *exit*;
  lKind ← *lLine*[1];
  if ¬(*lKind* ∈ *AvailableSymbols*) **then** *exit*

This code is used in section 558.

560. Recall the specification for *Val* sets *lCode* to zero for success, and the nonzero values store the index where the string is not a numeric value.

We copy the identifier (as determined from the start of the line until, but not including, the index of the first space in the line) and throw away everything after the first whitespace.

When the identifier for the functor symbol is not an empty string *and* the priority can be determined unambiguously, then the functor symbol entry is valid. Otherwise it is invalid.

⟨ Check if functor symbol is valid 560 ⟩ ≡
  begin *IsValidSymbol* ← *true*; *lPos* ← *Pos*(´␣´, *lLine*);
  if $lPos \neq 0$ **then**
    begin { Parse priority for symbol }
    val(*TrimString*(*Copy*(*lLine*, *lPos*, *length*(*lLine*))), *lPriority*, *lCode*);
    lLine ← *TrimString*(*Copy*(*lLine*, 1, $lPos - 1$)); *IsValidSymbol* ← $(lCode = 0) \land (lLine \neq$ ´´);
    end;
  end

This code is used in section 558.

561. A predicate entry in the dictionary file should not include a priority, nor should it include any whitespaces. This is the criteria for a valid predicate symbol entry in the dictionary.

We enforce this by finding the first "␣" character in the line. If there is one, then we trim both sides of the line (removing leading and trailing whitespace). We should have no more spaces in the line. If there is a space, then it is an invalid predicate symbol.

⟨ Check if predicate symbol is valid 561 ⟩ ≡
  begin *lPos* ← *Pos*(´␣´, *lLine*);
  if $lPos \neq 0$ **then** { *lLine* contains a space }
    begin *lLine* ← *TrimString*(*Copy*(*lLine*, *lPos*, *length*(*lLine*)));
    if *Pos*(´␣´, *lLine*) > 0 **then** *exit*;
    end;
  IsValidSymbol ← *true*;
  end

This code is used in section 558.

562. TSymbol. These are used in `kernel/accdict.pas`. The *Kind* is its one-letter kind (discussed in §555), and *Repr* is its lexeme. For functors, its priority is stored as its *Prior*.

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

⟨ Symbol for vocabulary 562 ⟩ ≡
  PSymbol = ↑*TSymbol*;
  TSymbol = **object** (*MObject*)
    Kind: *char*;
    Repr, *Infinitive*: *string*;
    Prior: *byte*;
    constructor *Init*(*fKind* : *char*; *fRepr*, *fInfinitive* : *string*; *fPriority* : *byte*);
    constructor *Extract*(**const** *aLine*: *string*);
    function *SymbolStr*: *string*;
    constructor *Load*(**var** *aText* : *text*);
    procedure *Store*(**var** *aText* : *text*);
    destructor *Done*; *virtual*;
  end

This code is used in section 556.

563. Constructor. Given the "kind", its "representation" and "infinitive", and its priority (as a number between 0 and 255), we can construct a symbol.

⟨ Implementation for `dicthan.pas` 558 ⟩ +≡
constructor *TSymbol.Init*(*fKind* : *char*; *fRepr*, *fInfinitive* : *string*; *fPriority* : *byte*);
  begin *Kind* ← *fKind*; *Repr* ← *fRepr*; *Prior* ← *fPriority*; *Infinitive* ← ´´;
  end;

564. Constructor. When we want to extract a symbol from a line in the dictionary file, care must be taken for functors (since they may contain an explicit priority) and for predicates. Predicates have an undocumented feature to allow "infinitives", so an acceptable predicate line in a dictionary may look like

```
Rpredicate infinitive
```

Although what Mizar does with infinitives, I do not know...

constructor *TSymbol.Extract*(**const** *aLine*: *string*);
  var *lPos*, *lCode*: *integer*; *lRepr*: *string*;
  begin *Kind* ← *aLine*[1]; *Repr* ← *TrimString*(*Copy*(*aLine*, 2, *length*(*aLine*))); *Prior* ← 0;
  Infinitive ← ´´;
  case *Kind* **of**
  ´O´: **begin** *lPos* ← *Pos*(´␣´, *Repr*); *Prior* ← *StandardPriority*;
    if *lPos* ≠ 0 **then** ⟨ Initialize explicit priority for functor entry in dictionary 566 ⟩;
    end;
  ´R´: **begin** *lPos* ← *Pos*(´␣´, *Repr*);
    if *lPos* ≠ 0 **then** ⟨ Initilize explicit infinitive for a predicate entry in dictionary 565 ⟩;
    end;
  endcases;
  end;

565. Predicates can have an optional infinitive, separated from the lexeme by a single whitespace. It remains unclear what Mizar uses predicate infinitives for, but it is a feature. This is written out to the `.vcx` file, according to `xml_dict.pas`.

Note that there are 4 predicates with infinitives in Mizar:
(1) `jumps_in` (infinitive: `jump_in`) occurs in the article `AMISTD_1`
(2) `halts_in` (infinitive: `halt_in`) occurs in the article `EXTPRO_1`
(3) `refers` (infinitive: `refer`) occurs in the article `SCMFSA7B`
(4) `destroys` (infinitive: `destroy`) occurs in the article `SCMFSA7B`

⟨ Initilize explicit infinitive for a predicate entry in dictionary 565 ⟩ ≡
  begin *lRepr* ← *Repr*; *Repr* ← ´´; *Repr* ← *TrimString*(*Copy*(*lRepr*, 1, *lPos* − 1));
  Infinitive ← *TrimString*(*Copy*(*lRepr*, *lPos* + 1, *length*(*lRepr*)));
  end

This code is used in section 564.

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

⟨ Initialize explicit priority for functor entry in dictionary 566 ⟩ ≡
  begin *lRepr* ← *Repr*; *Repr* ← ´´;
  val(*TrimString*(*Copy*(*lRepr*, *lPos* + 1, *length*(*lRepr*))), *Prior*, *lCode*); { Store the priority }
  Repr ← *TrimString*(*Copy*(*lRepr*, 1, *lPos* − 1)); { Store the lexeme }
  end

This code is used in section 564.

567. Serialize symbols. We can serialize a *TSymbol* object, which produces the sort of entry we'd expect to find in a dictionary. So we would have the symbol kind, the lexeme, and optional data (non-default priorities for functors, infinitives for predicates).

⟨ Implementation for `dicthan.pas` 558 ⟩ +≡

```
function TSymbol.SymbolStr: string;
  var lStr, lIntStr: string;
  begin lStr ← Kind + Repr;
  case Kind of
  ´O´: if Prior ≠ StandardPriority then
      begin Str(Prior, lIntStr); lStr ← lStr + ´␣´ + lIntStr;
      end;
  ´R´: if Infinitive ≠ ´´ then lStr ← lStr + ´␣´ + Infinitive;
  endcases;
  SymbolStr ← lStr;
  end;
```

568. Given a text (usually the contents of a vocabulary file), we read in a line. When the line is a nonempty string, we initialize the lexeme representation, priority, and infinitives. Then, when the dictionary entry describes a valid symbol (§558), we populate the fields of the *TSymbol*.

```
constructor TSymbol.Load(var aText : text);
  var lDictLine: string;
  begin ReadLn(aText, lDictLine); lDictLine ← TrimString(lDictLine);
  if length(lDictLine) = 0 then exit;
  Repr ← ´´; Prior ← 0; Infinitive ← ´´;
  if IsValidSymbol(lDictLine) then Extract(lDictLine);
  end;
```

569. Storing a *TSymbol* in a file amounts to writing its serialization (§567) to the file.

```
procedure TSymbol.Store(var aText : text);
  begin WriteLn(aText, SymbolStr);
  end;
```

570. Destructor. We just reset the lexeme and infinitive strings to be empty strings.

```
destructor TSymbol.Done;
  begin Repr ← ´´; Infinitive ← ´´;
  end;
```

571. Abstract vocabulary objects. This is used in `kernel/impobjs.pas`. We recall (§554) that the *SymbolCounters* are just an enumerated type consisting of a single uppercase Latin Letter.

⟨ Abstract vocabulary object declaration 571 ⟩ ≡

```
  AbsVocabularyPtr = ↑AbsVocabularyObj;
  AbsVocabularyObj = object (MObject)
    fSymbolCnt: SymbolCounters;
    constructor Init;
    destructor Done; virtual;
  end
```

This code is used in section 556.

572. We only have the constructor and destructor for abstract vocabulary objects.

⟨ Implementation for `dicthan.pas` 558 ⟩ +≡
constructor *AbsVocabularyObj.Init*;
  begin *FillChar*(*fSymbolCnt*, *SizeOf*(*fSymbolCnt*), 0);
  end;
destructor *AbsVocabularyObj.Done*;
  begin end;

573. Vocabulary objects. A "vocabulary object" is just a collection of *PSymbol*s read in from a vocabulary file.

These are also used in `kernel/accdict.pas`.

⟨ Vocabulary object declaration 573 ⟩ ≡
  PVocabulary = ↑*TVocabulary*;
  TVocabulary = **object** (*AbsVocabularyObj*)
    Reprs: *MCollection*;
    constructor *Init*;
    constructor *ReadPrivateVoc*(**const** *aFileName*: *string*);
    constructor *LoadVoc*(**var** *aText* : *text*);
    procedure *StoreVoc* (**const** *aFileName*: *string*; **var** *aText*: *text*) ;
    destructor *Done*; *virtual*;
  end

This code is used in section 556.

574. Constructor (Empty vocabulary). We can construct the empty vocabulary by just initializing the underlying collection.

⟨ Implementation for `dicthan.pas` 558 ⟩ +≡
constructor *TVocabulary.Init*;
  begin *FillChar*(*fSymbolCnt*, *SizeOf*(*fSymbolCnt*), 0); *Reprs.Init*(10, 10);
  end;

575. Destructor. We only need to free up the underlying collection.

destructor *TVocabulary.Done*;
  begin *Reprs.Done*;
  end;

576. 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 ¬*seekEOF*(*lDict*) **do** ⟨ Read line into vocabulary from dictionary file 577 ⟩;
  Close(*lDict*);
  end;

577. When reading dictionary lines into a vocabulary file, we skip over blank lines. Further, we only read *valid* entries into the vocabulary.

⟨Read line into vocabulary from dictionary file 577⟩ ≡
  begin *readln*(*lDict*, *lDictLine*); *lDictLine* ← *TrimString*(*lDictLine*);
  if *length*(*lDictLine*) > 1 **then** { if dictionary line is not blank }
    begin *lSymbol* ← *new*(*PSymbol*, *Extract*(*lDictLine*));
    if *IsValidSymbol*(*lDictLine*) **then** { add the symbol }
      begin *inc*(*fSymbolCnt*[*lSymbol*↑.*Kind*]); *Reprs*.*Insert*(*lSymbol*); **end**;
    end;
  end

This code is used in section 576.

578. 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.

⟨Implementation for `dicthan.pas` 558⟩ +≡
constructor *TVocabulary*.*LoadVoc*(**var** *aText* : *text*);
  var *i*, *lSymbNbr*, *lNbr*: *integer*; *lKind*, *lDummy*, *c*: *Char*;
  begin *lSymbNbr* ← 0; ⟨Count *lNbr* the number of dictionary entries for an article 579⟩;
  ReadLn(*aText*); *Reprs*.*Init*(10, 10);
  for *i* ← 1 **to** *lSymbNbr* **do**
    begin *Reprs*.*Insert*(*new*(*PSymbol*, *Load*(*aText*)));
    end;
  end;

579. 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*.

⟨Count *lNbr* the number of dictionary entries for an article 579⟩ ≡
  for *c* ← ´A´ **to** ´Z´ **do**
    if *c* ∈ *AvailableSymbols* **then**
      begin *Read*(*aText*, *lKind*, *lNbr*, *lDummy*); *fSymbolCnt*[*c*] ← *lNbr*; *Inc*(*lSymbNbr*, *fSymbolCnt*[*c*]);
      end

This code is used in section 578.

580. 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.

⟨Implementation for `dicthan.pas` 558⟩ +≡
procedure *TVocabulary*.*StoreVoc* (**const** *aFileName*: *string*; **var** *aText*: *text*) ;
  var *i*: *Byte*; *c*: *Char*;
    begin *WriteLn*(*aText*, ´#´, *aFileName*);
    for *c* ← ´A´ **to** ´Z´ **do**
      if *c* ∈ *AvailableSymbols* **then** *Write*(*aText*, *c*, *fSymbolCnt*[*c*], ´␣´);
    WriteLn(*aText*);
    for *i* ← 0 **to** *Reprs*.*Count* − 1 **do** *PSymbol*(*Reprs*.*Items*↑[*i*])↑.*Store*(*aText*);
    end;

581. Miscellaneous public-facing functions.

⟨ Implementation for `dicthan.pas` 558 ⟩ +≡

```
function GetPrivateVoc(const fName: string): PVocabulary;
  var lName: string;
  begin lName ← fName;
  if ExtractFileExt(lName) = ´´ then lName ← lName + ´.voc´;
  if ¬MFileExists(lName) then
    begin GetPrivateVoc ← nil; exit;
    end;
  GetPrivateVoc ← new(PVocabulary, ReadPrivateVoc(lName));
  end;
```

582. Reading mml.vct entries. The `$MIZFILES/mml.vct` file contains all the vocabularies concatenated together into one giant vocabulary file. It uses lines prefixed with "#" followed by the article name to separate the vocabularies from different files. We search for the given article name (stored in the *fName* argument). When we find it, we construct the Vocabulary object (§578).

```
function GetPublicVoc (const fName: string; var fVocFile: text ) : PVocabulary;
  var lLine: string;
    begin GetPublicVoc ← nil; reset(fVocFile);
    while ¬eof(fVocFile) do
      begin readln(fVocFile, lLine);
      if (length(lLIne) > 0) ∧ (lLine[1] = ´#´) ∧ (copy(lLine, 2, length(lLine)) = fName) then
        begin GetPublicVoc ← new(PVocabulary, LoadVoc(fVocFile)); exit;
        end;
      end;
    end;
```

583. Reading from mml.vct. This function is used by `libtools/checkvoc.dpr` and in a couple user tools. In those other functions, they pass `$MIZFILES/mml.vct` as the value for *aFileName*. This procedure will then populate the *aMmlVcb* file associating to each article name its vocabulary.

```
procedure LoadMmlVcb (const aFileName: string; var aMmlVcb: MStringList ) ;
  var lFile: text; lDummy: char; lDictName: string; r: Integer;
    begin FileExam(aFileName); Assign(lFile, aFileName); Reset(lFile);   { initialize file for reading }
    aMmlVcb.Init(1000); aMmlVcb.fSorted ← true;
    while ¬eof(lFile) do
      begin ReadLn(lFile, lDummy, lDictName);
      r ← aMmlVcb.AddObject(lDictName, new(PVocabulary, LoadVoc(lFile)));
      end;
    Close(lFile);
    end;
```

584. Storing a vocabulary delegates much work (§580). 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 ← 0 to fCount − 1 do PVocabulary(fList↑[i].fObject)↑.StoreVoc(fList↑[i].fString↑, lFile);
  Close(lFile);
  end;
```

585. Like *StoreMmlVcb*, this function is not used anywhere in Mizar. This appears to produce the XML-equivalent to the previous function.

```
procedure StoreMmlVcbX(const aFileName: string; const aMmlVcb: MStringList);
  var i, s: Integer; c: char; VCXfile: XMLOutStreamPtr;
  begin VCXfile ← new(XMLOutStreamPtr, OpenFile(aFileName));
  VCXfile.Out_XElStart0(XMLElemName[elVocabularies]);
  with aMmlVcb do
    for i ← 0 to fCount − 1 do
      with PVocabulary(fList↑[i].fObject)↑ do
        begin VCXfile.Out_XElStart(XMLElemName[elVocabulary]);
        VCXfile.Out_XAttr(XMLAttrName[atName], fList↑[i].fString↑); VCXfile.Out_XAttrEnd;
        ⟨Write vocabulary counts to XML file 586⟩;
        ⟨Write symbols to vocabulary XML file 587⟩;
        VCXfile.Out_XElEnd(XMLElemName[elVocabulary]);
        end;
  VCXfile.Out_XElEnd(XMLElemName[elVocabularies]); dispose(VCXfile, Done);
  end;
```

586. We write out the counts of each kind of definition appearing in the article.

```
⟨Write vocabulary counts to XML file 586⟩ ≡
    { Kinds }
  for c ← ´A´ to ´Z´ do
    if c ∈ AvailableSymbols then
      begin VCXfile.Out_XElStart(XMLElemName[elSymbolCount]);
      VCXfile.Out_XAttr(XMLAttrName[atKind], c);
      VCXfile.Out_XIntAttr(XMLAttrName[atNr], fSymbolCnt[c]); VCXfile.Out_XElEnd0;
      end
```

This code is used in section 585.

587. We write out each symbol appearing in the article's vocabulary.

```
⟨Write symbols to vocabulary XML file 587⟩ ≡
    { Symbols }
  VCXfile.Out_XElStart0(XMLElemName[elSymbols]);
  for s ← 0 to Reprs.Count − 1 do
    with PSymbol(Reprs.Items[s])↑ do
      begin VCXfile.Out_XElStart(XMLElemName[elSymbol]);
      VCXfile.Out_XAttr(XMLAttrName[atKind], Kind);
      VCXfile.Out_XAttr(XMLAttrName[atName], QuoteStrForXML(Repr));
      case Kind of
      ´O´: VCXfile.Out_XIntAttr(XMLAttrName[atPriority], Prior);
      ´R´: if Infinitive ≠ ´´ then VCXfile.Out_XAttr(XMLAttrName[atInfinitive], Infinitive);
      end; VCXfile.Out_XElEnd0;
      end;
  VCXfile.Out_XElEnd(XMLElemName[elSymbols])
```

This code is used in section 585.

File 16

Scanner

588. The `scanner.pas` file contains the *MTokeniser* and the *MScanner*.

It is worth noting: if we want to extend Mizar to support Unicode, then we would want to hack this file accordingly. Or create a `utf8scanner` module, whichever. This scanner class is built specifically to work with ASCII characters, specifically accepting printable characters and the space ("␣") characters as valid input.

⟨scanner.pas 588⟩ ≡
  ⟨GNU License 4⟩
unit *scanner*;
  interface ;
  uses *errhan*, *mobjects*;
  const *MaxLineLength* = 80;
    MaxConstInt = 2147483647; { $= 2^{31} - 1$, maximal signed 32-bit integer }
  ⟨Type declarations for scanner 589⟩
  implementation
  uses *mizenv*, *librenv*, *mconsole*, *xml_dict*, *xml_inout*;
  ⟨Implementation for scanner.pas 590⟩
  end .

See also section 719.

589. Note that a *LexemRec* is really a standardized token. I was always raised to believe that a "lexeme" refers to the literal text underlying a token.

⟨Type declarations for scanner 589⟩ ≡
type *ASCIIArr* = **array** [*chr*(0) . . *chr*(255)] **of** *byte*;
  LexemRec = **record** *Kind*: *char*;
    Nr: *integer*;
    end;
  ⟨Token object class 591⟩;
  ⟨Tokens collection class 593⟩;
  ⟨MToken object class 601⟩;
  ⟨MTokeniser class 604⟩;
  ⟨MScanner object class 632⟩;

This code is used in section 588.

590. The "default allowed" characters are the 10 decimal digits, the 26 uppercase Latin letters, the 26 lowercase Latin letters, and the underscore ("_") character.

⟨ Implementation for scanner.pas 590 ⟩ ≡
var *DefaultAllowed*: *AsciiArr* =
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0, 0, 0,
0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1,
{ '_' allowed in identifiers by default! }
0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);

See also sections 592, 594, 595, 596, 599, 600, 602, 603, 605, 608, 609, 613, 628, 629, 631, 633, 642, 643, 644, 645, and 646.

This code is used in section 588.

591. Tokens object. A token contains a lexeme, but it extends an *MStr* object.

⟨ Token object class 591 ⟩ ≡
TokenPtr = ↑*TokenObj*;
TokenObj = **object** (*MStrObj*)
fLexem: *LexemRec*;
constructor *Init*(*aKind* : *char*; *aNr* : *integer* ; **const** *aSpelling*: *string*);
end

This code is used in section 589.

592. The constructor for a token requires its kind (functor, mode, predicate, etc.), and its internal "number", as well as its raw lexeme *aSpelling*.

⟨ Implementation for scanner.pas 590 ⟩ +≡
constructor *TokenObj.Init*(*aKind* : *char*; *aNr* : *integer* ; **const** *aSpelling*: *string*);
begin *fLexem.Kind* ← *aKind*; *fLexem.Nr* ← *aNr*; *fStr* ← *aSpelling*;
end;

Section 16.1. COLLECTIONS OF TOKENS

593. We can populate a token collection from a dictionary file, or we can start with an empty collection. We can save our collection to a file. We can also insert (or "collect") a new token into the collection.

⟨ Tokens collection class 593 ⟩ ≡
  TokensCollection = **object** (*MSortedStrList*)
    fFirstChar: **array** [*chr*(30) .. *chr*(255)] **of** *integer*;
    constructor *InitTokens*;
    constructor *LoadDct*(**const** *aDctFileName*: *string*);
    procedure *SaveDct*(**const** *aDctFileName*: *string*);
    procedure *SaveXDct*(**const** *aDctFileName*: *string*);
    function *CollectToken*(**const** *aLexem*: *LexemRec*; **const** *aSpelling*: *string*): *boolean*;
  end

This code is used in section 589.

594. Construct empty token collection.

⟨ Implementation for scanner.pas 590 ⟩ +≡
constructor *TokensCollection.InitTokens*;
  begin *Init*(100);
  end;

595. Insert. If the collection already contains the token described by *aLexem*, then we just free up the memory allocated for the token (avoid duplicates). Otherwise, we insert the token.

⟨ Implementation for scanner.pas 590 ⟩ +≡
function *TokensCollection.CollectToken*(**const** *aLexem*: *LexemRec*; **const** *aSpelling*: *string*): *boolean*;
  var *k*: *integer*; *lToken*: *TokenPtr*;
  begin *lToken* ← *new*(*TokenPtr*, *Init*(*aLexem.Kind*, *aLexem.Nr*, *aSpelling*));
  if *Search*(*lToken*, *k*) **then** { already contains token? }
    begin *CollectToken* ← *false*; *dispose*(*lToken*, *Done*)
    end
  else begin *CollectToken* ← *true*; *Insert*(*lToken*)
    end
  end;

596. Load a dictionary. We open the dictionary "`.dct`" file (expects the file name to be lacking that extension), and construct an empty token collection. Then we iterate through the dictionary, reading each line, forming a new token, then inserting it into the collection.

The "`.dct`" file contains all the identifiers from articles referenced in the `environ` part of an article, and it will always have the first 148 lines be for reserved keywords. The format for a "`.dct`" file consists of lines of the form

$$\langle kind\rangle\langle number\rangle␣\langle name\rangle$$

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.

⟨ Implementation for scanner.pas 590 ⟩ +≡
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*;
  ⟨ Load all tokens from the dictionary 597 ⟩;
  close(*Dct*); ⟨ Index first character appearances among definitions 598 ⟩;
  end;

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

⟨ Load all tokens from the dictionary 597 ⟩ ≡
  while ¬*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 596.

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

⟨ Index first character appearances among definitions 598 ⟩ ≡
  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*↑[*fIndex*↑[*i*]])↑.*fStr*[1];
    if $fFirstChar[c] = -1$ **then** $fFirstChar[c] \leftarrow i$;
    end

This code is used in section 596.

599. We save a token collection to a "`.dct`" file. This appears to just produce the concatenation of the definition kind, the identifier number, then a whitespace separating it from the lexeme. **Caution:** this is *not* an XML format! For that, see *SaveDctX*.

⟨ Implementation for scanner.pas 590 ⟩ +≡
procedure *TokensCollection*.*SaveDct*(**const** *aDctFileName*: *string*);
  var *i*: *integer*; *DctFile*: *text*;
  begin *assign*(*DctFile*, *aDctFileName* + ´`.dct`´); *rewrite*(*DctFile*);
  for $i \leftarrow 0$ **to** $Count - 1$ **do**
    with *TokenPtr*(*Items*↑[*i*])↑, *fLexem* **do** *writeln*(*DctFile*, *AnsiChar*(*Kind*), *Nr*, ´␣´, *fStr*);
  close(*DctFile*);
  end;

600. Save dictionary to XML file. The RNC (compact Relax NG Schema): Local dictionary for an article. The symbol kinds still use very internal notation.

```
elSymbols =
attribute atAid {xsd:string}?,
element elSymbols {
  element elSymbol {
    attribute atKind {xsd:string},
    attribute atNr {xsd:integer},
    attribute atName {xsd:integer}
  }*
}
```

This creates the `.dct` file for an article.

⟨ Implementation for scanner.pas 590 ⟩ +≡

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*↑[*i*])↑, *fLexem* **do**
        begin *Out_XElStart*(*XMLElemName*[*elSymbol*]);
        Out_XQuotedAttr(*XMLAttrName*[*atKind*], *Kind*); *Out_XIntAttr*(*XMLAttrName*[*atNr*], *Nr*);
        Out_XQuotedAttr(*XMLAttrName*[*atName*], *fStr*); *Out_XElEnd0*;
        end;
    Out_XElEnd(*XMLElemName*[*elSymbols*]); { print `elSymbols` end-tag }
    end;
  lEnvFile.Done;
  end;

Section 16.2. MIZAR TOKEN OBJECTS

601. This appears to be tokens for a specific file. An MToken extends a Token (§591).

⟨ MToken object class 601 ⟩ ≡
  MTokenPtr = ↑*MTokenObj*;
  MTokenObj = **object** (*TokenObj*)
    fPos: *Position*;
    constructor *Init*(*aKind* : *char*; *aNr* : *integer* ; **const** *aSpelling*: *string*; **const** *aPos*: *Position*);
  end

This code is used in section 589.

602. Constructor. Construct a token. This might be a tad confusing, at least for me, because the lexeme is stored in the *fStr* field, whereas the standardized token is stored in the *fLexem* field.

We do not need to invoke the constructor for any ancestor class, because we just construct everything here. This seems like a bug waiting to happen. . .

⟨ Implementation for scanner.pas 590 ⟩ +≡
constructor *MTokenObj.Init*(*aKind* : *char*; *aNr* : *integer*;
    const *aSpelling*: *string*;
    const *aPos*: *Position*);
  begin *fLexem.Kind* ← *aKind*; *fLexem.Nr* ← *aNr*; *fStr* ← *aSpelling*; *fPos* ← *aPos*;
  end;

603. Token Kind constants. There are four kinds of tokens we want to distinguish: all valid tokens are either (1) numerals, or (2) identifiers. Then we also have (3) error tokens. But last, we have (4) end of text tokens.

These are for identifying everything which is neither an identifier defined in the vocabulary files, nor a reserved keyword.

⟨ Implementation for scanner.pas 590 ⟩ +≡
const *Numeral* = ´N´; *Identifier* = ´I´; *ErrorSymbol* = ´?´; *EOT* = ´!´;

Section 16.3. TOKENISER

604. The first step in lexical analysis is to transform a character stream into a token stream. The Tokeniser extends the MToken object (§601), which in turn extends the Token object (§591).

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 ("␣") character or it will be the empty string. Any class extending *MTokeniser* must respect this contract.

⟨ MTokeniser class 604 ⟩ ≡
  MTokeniser = **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** *aToken*: *LexemRec*): *string*; *virtual*;
  end

This code is used in section 589.

605. Spelling boils down to three cases (c.f., types of tokens §603): numerals, identifiers, and everything else. Numerals spell out the base-10 decimal expansion.

The other two cases boil down to finding the first matching token in the caller's collection of tokens with the same lexeme supplied as an argument, provided certain 'consistency' checks hold (the lexeme and token have the same *Kind*).

⟨ Implementation for scanner.pas 590 ⟩ +≡
function *MTokeniser.Spelling*(**const** *aToken*: *LexemRec*): *string*;
  var *i*: *integer*; *s*: *string*;
  begin *Spelling* ← ´´;
  if *aToken.Kind* = *Numeral* **then**
    begin *Str*(*aToken.Nr*, *s*); *Spelling* ← *s*; **end**
  else if *aToken.Kind* = *Identifier* **then** ⟨ Spell an identifier for the MTokeniser 606 ⟩
    else ⟨ Spell an error or EOF for the MTokeniser 607 ⟩;
  end;

606. Spelling an identifier just needs to match the lexeme's number with the token's number. This finds the first matching token in the underlying collection, then terminates the function.

⟨Spell an identifier for the MTokeniser 606⟩ ≡
  begin for $i \leftarrow 0$ **to** *fIdents.Count* $- 1$ **do**
    with *TokenPtr*(*fIdents.Items*↑[*i*])↑ **do**
      if *fLexem.Nr* = *aToken.Nr* **then**
        begin *Spelling* ← *fStr*; *exit*
        end;
  end

This code is used in section 605.

607. Spelling anything else for the tokeniser needs the kind and number of the lexeme to match those of the token. Again, this finds the first matching token in the underlying collection, then terminates the function.

⟨Spell an error or EOF for the MTokeniser 607⟩ ≡
  begin for $i \leftarrow 0$ **to** *fTokens.Count* $- 1$ **do**
    with *TokenPtr*(*fTokens.Items*↑[*i*])↑ **do**
      if (*fLexem.Kind* = *aToken.Kind*) ∧ (*fLexem.Nr* = *aToken.Nr*) **then**
        begin *Spelling* ← *fStr*; *exit*
        end;
  end

This code is used in section 605.

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

⟨Implementation for scanner.pas 590⟩ +≡
constructor *MTokeniser.Init*;
  begin *fPos.Line* ← 0; *fLexem.Kind* ← ´␣´; *fPhrase* ← ´␣␣´; *fPhrasePos.Line* ← 0;
  fPhrasePos.Col ← 0; *fTokensBuf.Init*(80, 8); *fTokens.Init*(0); *fIdents.Init*(100);
  end;

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

⟨Implementation for scanner.pas 590⟩ +≡
destructor *MTokeniser.Done*;
  begin *fPhrase* ← ´´; *fTokensBuf.Done*; *fTokens.Done*; *fIdents.Done*;
  end;

610. Aside on ASCII separators. Note: *chr*(30) is the record separator in ASCII, and *chr*(31) is the unit separator. Within a group (or table), the records are separated with the "RS" (*chr*(30)). As far as unit separators, Lammer Bies explains (`lammertbies.nl/comm/info/ascii-characters`):

> The smallest data items to be stored in a database are called units in the ASCII definition. We would call them field now. The unit separator separates these fields in a serial data storage environment. Most current database implementations require that fields of most types have a fixed length. Enough space in the record is allocated to store the largest possible member of each field, even if this is not necessary in most cases. This costs a large amount of space in many situations. The US control code allows all fields to have a variable length. If data storage space is limited—as in the sixties—this is a good way to preserve valuable space. On the other hand is serial storage far less efficient than the table driven RAM and disk implementations of modern times. I can't imagine a situation where modern SQL databases are run with the data stored on paper tape or magnetic reels. . .

We will introduce macros for the record separator and the unit separator, because Mizar's front-end uses them specifically for the following purposes:

(1) lines longer than 80 characters will contain a *record_separator* character (§637);
(2) all other invalid characters are replaced with the *unit_separator* character (c.f., §638).

define *record_separator* ≡ *chr*(30)
define *unit_separator* ≡ *chr*(31)

611. Example of zeroeth step ("getting a phrase") in tokenising. The *GetPhrase* function is left as an abstract method of the tokeniser, so it is worth discussing "What it is supposed to do" before getting to the tokenisation of strings.

Suppose we have the following snippet of Mizar:

```
begin

theorem
  for x being object
  holds x= x;
```

This is "sliced up" into the following "phrases" (drawn in boxes) which are clustered by lines:

```
[begin␣]
[theorem␣]
  [for␣] [x␣] [being␣] [object␣]
  [holds␣] [x=␣] [x;␣]
```

Observe that the "phrases" are demarcated by whitespaces ("␣") 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=␣`" contains the two tokens "`x`" and "`=`").

What is the contract for a "phrase"? A phrase is *guaranteed* to either be equal to "␣", or it contains at least one token and it is *guaranteed* to end with a space "␣" character (ASCII code `#32`). Further, there are no other possible "␣" 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.

612. Tokenise a phrase. When a "phrase" has been loaded into the tokeniser (which is an abstract method implemented by its descendent classes), we tokenise it — "slice it up" into tokens, thereby populating the *fTokensBuf* tokens buffer. This is invoked as needed by the *GetToken* method (§629).

This function is superficially complex, but upon closer scrutiny it is fairly straightforward.

Also note, despite being marked as "virtual", this is not overridden anywhere in the Mizar program.

The contract ensures, barring catastrophe, the *fLexem*, *fStr*, and *fPos* be populated. **Importantly:** The *fLexem*'s token type is one of the four kinds given in the constant section (§603): `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 (§619) in this procedure.

⟨Variables for slicing a phrase 612⟩ ≡
lCurrChar: *integer*; { index in *fPhrase* for current position }
EndOfSymbol: *integer*;
EndOfIdent: *integer*; { index in *fPhrase* for end of identifier }
FoundToken: *TokenPtr*; { most recently found token temporary variable }
lPos: *Position*; { position for debugging purposes }

See also sections 615, 618, 621, 623, and 625.

This code is used in section 613.

613. ⟨Implementation for scanner.pas 590⟩ +≡
procedure *MTokeniser.SliceIt*;
  var ⟨Variables for slicing a phrase 612⟩
  begin *MizAssert*(2333, *fTokensBuf.Count* = 0); { Requires: token buffer is empty }
  lCurrChar ← 1; *lPos* ← *fPhrasePos*;
  ⟨Slice pragmas 614⟩;
  while *fPhrase*[*lCurrChar*] ≠ ´␣´ **do**
    begin ⟨Determine the ID 616⟩;
    ⟨Try to find a dictionary symbol 619⟩;
    if *EndOfSymbol* < *EndOfIdent* **then** ⟨Check identifier is not a number 622⟩;
    if *FoundToken* ≠ **nil then**
      with *FoundToken*↑ **do**
        begin *lPos.Col* ← *fPhrasePos.Col* + *EndOfSymbol* − 1;
        fTokensBuf.Insert(*new*(*MTokenPtr*, *Init*(*fLexem.Kind*, *fLexem.Nr*, *fStr*, *lPos*)));
        lCurrChar ← *EndOfSymbol* + 1; *continue*;
        end;
    { else *FoundToken* = **nil** }
    ⟨Whoops! We found an unknown token, insert a 203 error token 627⟩;
    end;
  end;

614. We begin by slicing pragmas. This will insert the pragma into the tokens buffer.

Note that the "`$EOF`" pragma indicates that we should treat the file as ending here. So we comply with the request, inserting the *EOT* (end of text) token as the next token to be offered to the user.

⟨Slice pragmas 614⟩ ≡
  if (*lPos.Col* = 1) ∧ (*Pos*(´`::$`´, *fPhrase*) = 1) **then**
    begin *fTokensBuf.Insert*(*new*(*MTokenPtr*, *Init*(´␣´, 0, *copy*(*fPhrase*, 3, *length*(*fPhrase*) − 3), *lPos*)));
    if *copy*(*fPhrase*, 1, 6) = ´`::$EOF`´ **then**
      fTokensBuf.Insert(*new*(*MTokenPtr*, *Init*(*EOT*, 0, *fPhrase*, *lPos*)));
    exit
    end

This code is used in section 613.

615. We take the longest possible substring consisting of identifier characters as a possible identifier. The phrase is guaranteed to contain at least one token, maybe more, so we just keep going until we have exhausted the phrase or found a non-identifier character.

Note that all invalid characters are transformed into the "unit character" (c.f., §638). We should treat any occurrence of them as an error.

At the end of this stage of our tokenising journey, for valid tokens, we should have *EndOfIdent* and *IdentLength* both initialized here.

⟨ Variables for slicing a phrase 612 ⟩ +≡
IdentLength: *integer*;

616. ⟨ Determine the ID 616 ⟩ ≡
{ 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 617 ⟩;
dec(*EndOfIdent*)

This code is used in section 613.

617. Recall (§637), we treat record separators as indicating the line is "too long" (i.e., more than 80 characters long). So we insert a 201 "Too long source line" error. But anything else is treated as an invalid identifier error.

⟨ Whoops! ID turns out to be invalid, insert an error token, then continue 617 ⟩ ≡
begin *lPos.Col* ← *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 ← *EndOfIdent* + 1; *continue*;
end

This code is used in section 616.

618. We look at the current phrase and try to match against tokens found in the underlying dictionary. When we find a match, we check if there are *multiple* matches and return the last one (this reflects Mizar's "the last version of the notation is preferred"). We implement this matching scheme using an infinite loop. Note that this uses a "**repeat**... **until** *false*" loop, which is identical to "**while** *true* **do begin** ... **end**" loop. (I am tempted to introduce a macro just to have this loop "**repeat**... **until** *end_of_time*"...)

Recall (§250), sorted lists have a field *fIndex* which is an array of indices (which are sorted while leaving the underlying array *Items* of data untouched).

Also, *lToken*, *lIndex* are used only in this code chunk. Here *lToken* is translated to an index of the underlying dictionary, so for clarity we introduce a macro to refer to the token directly. And *lIndex* is used as "the current character" index to compare the phrase to the token (indexed by *lToken*) as a match or not.

At the end of this chunk, if successful, then *FoundToken* will be set to a valid token pointer. Further, *EndOfPhrase* will be initialized.

A possible bug: what happens if we look through the entire phrase? We can't "look any farther" down the phrase, so shouldn't we throw an error? Or lazily read more characters? Or... something?

Never fear: this will never happen with Mizar's grammar. The "reserved words" are *always* separated from the other stuff by at least one whitespace.

Also we note the list of symbols is sorted lexicographically.

This appears to match the phrase with the longest possible matching entry in the list of symbols (it is "maximal munch").

define *the_item*(#) ≡ *Items*↑[*fIndex*↑[#]]
define *the_token*(#) ≡ *TokenPtr*(*the_item*(#))↑

⟨ Variables for slicing a phrase 612 ⟩ +≡
EndOfPhrase: *integer*; { index in *fPhrase* for candidate token }
lIndex: *integer*; { index for *fIndex* entry }
lToken: *integer*; { index for entries in dictionary starting with the first character of the current token }

619. Reserved keywords and defined terms are loaded into the *fTokens* dictionary.

⟨ Try to find a dictionary symbol 619 ⟩ ≡
```
EndOfPhrase ← lCurrChar; FoundToken ← nil; EndOfSymbol ← EndOfPhrase − 1;
      { initialized for comparison }
  lToken ← fTokens.fFirstChar[fPhrase[EndOfPhrase]]; inc(EndOfPhrase);
  if (lToken ≥ 0) then
    with fTokens do
      begin lIndex ← 2;
      repeat   { infinite loop }
        ⟨ If we matched a dictionary entry, then initialize FoundToken 620 ⟩;
        if fPhrase[EndOfPhrase] = ´␣´ then break;   { we are done! }
        if (lIndex ≤ length(the_token(lToken).fStr)) ∧
              (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 613.

620. 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*.

⟨ If we matched a dictionary entry, then initialize *FoundToken* 620 ⟩ ≡
  if *lIndex* > *length*(*the_token*(*lToken*).*fStr*) **then** { we matched the token }
    begin *FoundToken* ← *the_item*(*lToken*); *EndOfSymbol* ← *EndOfPhrase* − 1;
    end

This code is used in section 619.

621. When the identifier is not a number, we insert an "identifier" token into the tokens buffer.

⟨ Variables for slicing a phrase 612 ⟩ +≡
lFailed: *integer*; { index of first non-digit character }
I: *integer*; { index ranging over the raw lexeme string }
lSpelling: *string*; { raw lexeme as a string }

622. ⟨ Check identifier is not a number 622 ⟩ ≡
  begin *lSpelling* ← *copy*(*fPhrase*, *lCurrChar*, *IdentLength*);
  lPos.*Col* ← *fPhrasePos*.*Col* + *EndOfIdent* − 1;
  if (*ord*(*fPhrase*[*lCurrChar*]) > *ord*(´0´)) ∧ (*ord*(*fPhrase*[*lCurrChar*]) ≤ *ord*(´9´)) **then**
    begin *lFailed* ← 0; { location of non-digit character }
    for *I* ← 1 **to** *IdentLength* − 1 **do**
      if (*ord*(*fPhrase*[*lCurrChar* + *I*]) < *ord*(´0´)) ∨ (*ord*(*fPhrase*[*lCurrChar* + *I*]) > *ord*(´9´)) **then**
        begin *lFailed* ← *I* + 1; *break*;
        end;
    if *lFailed* = 0 **then** { if all characters are digits }
      ⟨ Whoops! Identifier turned out to be a number! 626 ⟩;
    end;
  ⟨ Add token to tokens buffer and iterate 624 ⟩;
  end

This code is used in section 613.

623. We add an *Identifier* token to the tokens buffer.

⟨ Variables for slicing a phrase 612 ⟩ +≡
lIdent: *TokenPtr*;

624. ⟨ Add token to tokens buffer and iterate 624 ⟩ ≡
  lIdent ← *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*↑[*I*])↑.*fLexem*.*Nr*, *lSpelling*, *lPos*)));
  lCurrChar ← *EndOfIdent* + 1; *continue*

This code is used in section 622.

625. If we goofed and all the characters turned out to be digits (i.e., the identifier *was* a numeral after all), we should clean things up here. Observe we will end up *continue*-ing along the loop.

When the numeral token is larger than $MaxConstInt = 2^{31} - 1$ (the largest 32-bit integer, §588), then we should raise a "Too large numeral" 202 error token. If we wanted to support "arbitrary precision" numbers, then this should be modified.

We can either insert into the tokens buffer an error token (in two possible outcomes) or a numeral token (in the third possible outcome).

⟨ Variables for slicing a phrase 612 ⟩ +≡
lNumber: *longint*;
J: *integer*;

626. ⟨ Whoops! Identifier turned out to be a number! 626 ⟩ ≡

```
begin if IdentLength > length(IntToStr(MaxConstInt)) then   { insert error token }
  begin fTokensBuf.Insert(new(MTokenPtr, Init(ErrorSymbol, 202, lSpelling, lPos)));
  lCurrChar ← EndOfIdent + 1; continue;
  end;
lNumber ← 0; J ← 1;
for I ← IdentLength − 1 downto 0 do
  begin lNumber ← lNumber + (ord(fPhrase[lCurrChar + I]) − ord('0')) ∗ J; J ← J ∗ 10;
  end;
if lNumber > MaxConstInt then   { insert error token }
  begin fTokensBuf.Insert(new(MTokenPtr, Init(ErrorSymbol, 202, lSpelling, lPos)));
  lCurrChar ← EndOfIdent + 1; continue;
  end;   { insert numeral token }
fTokensBuf.Insert(new(MTokenPtr, Init(Numeral, lNumber, lSpelling, lPos)));
lCurrChar ← EndOfIdent + 1; continue;
end
```

This code is used in section 622.

627. If we have tokenised the phrase, but the token is not contained in the dictionary, then we should raise a 203 error.

⟨ Whoops! We found an unknown token, insert a 203 error token 627 ⟩ ≡

```
lPos.Col ← fPhrasePos.Col + lCurrChar − 1;
fTokensBuf.Insert(new(MTokenPtr, Init(ErrorSymbol, 203, fPhrase[lCurrChar], lPos))); inc(lCurrChar)
```

This code is used in section 613.

628. We have purely abstract methods which will invoke *Abstract1* (§183), which raises a runtime error.

⟨ Implementation for scanner.pas 590 ⟩ +≡

```
procedure MTokeniser.GetPhrase;
  begin Abstract1;
  end;
function MTokeniser.EndOfText: boolean;
  begin Abstract1; EndOfText ← false;
  end;
function MTokeniser.IsIdentifierLetter(ch : char): boolean;
  begin Abstract1; IsIdentifierLetter ← false;
  end;
```

629. Get a token. Getting a token from the tokeniser will check if we've exhausted the input stream (which tests if the kind of *fLexem* is *EOT*), and exit if we have.

Otherwise, it looks to see if we've got tokens left in the buffer. If so, just pop one and exit.

But when the token buffer is empty, we invoke the abstract method *GetPhrase* to read some of the input stream. If it turns out there's nothing left to read, then update the tokeniser to be in the "end of text" state.

When we have some of the input stream read into the *fPhrase* field, we tokenise it using the *SliceIt* function. Then we pop a token from the buffer of tokens.

This will populate *fLexem*, *fStr*, and *fPos* with the new token, lexeme, and position... but that's only because *GetPhrase* (§633) and *SliceIt* (§612) do the actual work.

```
⟨Implementation for scanner.pas 590⟩ +≡
procedure MTokeniser.GetToken;
  begin if fLexem.Kind = EOT then exit;
  if fTokensBuf.Count > 0 then
    begin ⟨Pop a token from the underlying tokens stack 630⟩;
    exit;
    end;
  GetPhrase;
  if EndOfText then
    begin fLexem.Kind ← EOT; fStr ← ´´; fPos ← fPhrasePos; inc(fPos.Col);
    exit;
    end;
  SliceIt; ⟨Pop a token from the underlying tokens stack 630⟩;
  end;
```

630. Popping a token will update the lexeme, str, and position fields to be populated from the first item in the tokens buffer. Then it will free that item from the tokens buffer, shifting everything down by one.

```
⟨Pop a token from the underlying tokens stack 630⟩ ≡
  fLexem ← MTokenPtr(fTokensBuf.Items↑[0])↑.fLexem;
  fStr ← MTokenPtr(fTokensBuf.Items↑[0])↑.fStr; fPos ← MTokenPtr(fTokensBuf.Items↑[0])↑.fPos;
  fTokensBuf.AtFree(0)
```

This code is used in sections 629 and 629.

631. Testing if the given character is an identifier character or not requires invoking the abstract method *IsIdentifierLetter* (§628).

```
⟨Implementation for scanner.pas 590⟩ +≡
function MTokeniser.IsIdentifierFirstLetter(ch : char): boolean;
  begin IsIdentifierFirstLetter ← IsIdentifierLetter(ch);
  end;
```

Section 16.4. SCANNER OBJECT

632. This extends the Tokeniser class (§604). It is the only class extending the Tokeniser class.

⟨ MScanner object class 632 ⟩ ≡
  MScannPtr = ↑*MScannObj*;
  MScannObj = **object** (*MTokeniser*)
    Allowed: *ASCIIArr*;
    fSourceBuff : *pointer*;
    fSourceBuffSize: *word*;
    fSourceFile: *text*;
    fCurrentLine: *string*;
    constructor *InitScanning*(**const** *aFileName*, *aDctFileName*: *string*);
    destructor *Done*; *virtual*;
    procedure *GetPhrase*; *virtual*;
    procedure *ProcessComment*(*fLine*, *fStart* : *integer*; *cmt* : *string*); *virtual*;
    function *EndOfText*: *boolean*; *virtual*;
    function *IsIdentifierLetter*(*ch* : *char*): *boolean*; *virtual*;
  end

This code is used in section 589.

633. Get a phrase. We search through the lines for the "first phrase" (i.e., first non-whitespace character, which indicates the start of something interesting). Comments are thrown away as are Mizar pragmas.

This will update *fCurrentLine* as needed, setting it to the next line in the input stream buffer. It will assign a *copy* of the phrase to the field *fPhrase*, as well as update the *fPhrasePos*.

There is a comment in Polish, "uzyskanie pierwszego znaczacego znaku", which Google translates as: "obtaining the first significant sign". This seemed like a natural "chunk" of code to study in isolation.

The contract for *GetPhrase* ensures the *fPhrase* will be populated with a string ending with a space ("␣") character, or it will be the empty string (when the end of text has been encountered).

⟨ Implementation for scanner.pas 590 ⟩ +≡
procedure *MScannObj.GetPhrase*;
  const *Prohibited*: *ASCIIArr* = ⟨ Characters prohibited by *MScanner* 634 ⟩;
  var *i*, *k*: *integer*;
  begin *fPhrasePos.Col* ← *fPhrasePos.Col* + *length*(*fPhrase*) − 1;
  ⟨ Find the first significant 'sign' 635 ⟩;
  for *i* ← *fPhrasePos.Col* **to** *length*(*fCurrentLine*) **do**
    if *fCurrentLine*[*i*] = ´␣´ **then** *break*;
  fPhrase ← *Copy*(*fCurrentLine*, *fPhrasePos.Col*, *i* − *fPhrasePos.Col* + 1);
  end;

634. The prohibited ASCII characters are everything *NOT* among the follow characters:

```
␣ ! " # $ % & ' ( ) * + , - . / : ; < = > ? @
[ \ ] ^ _ ` { | } ~ 0 1 2 3 4 5 6 7 8 9
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

The reader will observe these are all the "graphic" ASCII characters, plus the space ("␣") character.

⟨Characters prohibited by *MScanner* 634⟩ ≡
(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)

This code is used in section 633.

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

⟨Find the first significant 'sign' 635⟩ ≡
while *fCurrentLine*[*fPhrasePos.Col*] = ´␣´ **do**
  begin if *fPhrasePos.Col* ≥ *length*(*fCurrentLine*) **then** ⟨Populate the current line 636⟩;
  inc(*fPhrasePos.Col*);
  end

This code is used in section 633.

636. Now, populating the current line requires a bit of work. We ensure the end of the current line will end with a space character ("␣"), which will guarantee the loop iteratively consumes all empty lines in the file.

Once we arrive at a non-space character, we will break the loop containing this chunk of code. If we have exhausted the underlying input stream, then we will have *EndOfText* be true. Should that occur, we exit the function.

⟨Populate the current line 636⟩ ≡
begin if *EndOfText* **then** *exit*;
inc(*fPos.Line*); *inc*(*fPhrasePos.Line*); *readln*(*fSourceFile*, *fCurrentLine*);
⟨Scan for pragmas, and exit if we found one 640⟩;
⟨Skip comments 641⟩;
⟨Trim whitespace from the right of the current line 639⟩;
⟨Replace every invalid character in current line with the unit character 638⟩;
fCurrentLine ← *fCurrentLine* + ´␣´;
if ¬*LongLines* **then**
  if *length*(*fCurrentLine*) > *MaxLineLength* **then** ⟨Replace end of long line with record separator 637⟩;
{ Assert: we have *fCurrentLine* end in "␣" }
fPhrasePos.Col ← 0; *fPos.Col* ← 0;
end

This code is used in section 635.

637. When we have excessively long lines, and we have not enabled "long line mode", then we just delete everything after $MaxLineLength + 1$ and set $MaxLineLength - 1$ to the record separator (which is rejected by the Mizar lexer) and the last character in the line to the space character.

⟨ Replace end of long line with record separator 637 ⟩ ≡
```
begin delete(fCurrentLine, MaxLineLength + 1, length(fCurrentLine));
fCurrentLine[MaxLineLength − 1] ← record_separator;
fCurrentLine[MaxLineLength] ← ´␣´;
end
```
This code is used in section 636.

638. In particular, if we every encounter an "invalid" character, then we just replace it with the "unit separator" character.

⟨ Replace every invalid character in current line with the unit character 638 ⟩ ≡
```
for k ← 1 to length(fCurrentLine) − 1 do
  if Prohibited[fCurrentLine[k]] > 0 then fCurrentLine[k] ← unit_separator
```
This code is used in section 636.

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

⟨ Trim whitespace from the right of the current line 639 ⟩ ≡
```
k ← length(fCurrentLine);
while (k > 0) ∧ (fCurrentLine[k] = ´␣´) do dec(k);
delete(fCurrentLine, k + 1, length(fCurrentLine))
```
This code is used in sections 636 and 640.

640. Pragmas in Mizar are special comments which start a line with "`::$`". They are useful for naming theorems ("`::$N` ⟨*name*⟩"), or toggling certain phases of the Mizar checker. This will process the comment (§642).

Since pragmas are important, we treat it as a token (and not a comment to be thrown away).

Note: if you try to invoke a pragma, but do not place it at the start of a line, then Mizar will treat it like a comment.

⟨ Scan for pragmas, and exit if we found one 640 ⟩ ≡
```
k ← Pos(´::$´, fCurrentLine);   { Preprocessing directive }
if (k = 1) then
  begin ProcessComment(fPhrasePos.Line, 1, copy(fCurrentLine, 1, length(fCurrentLine)));
  ⟨ Trim whitespace from the right of the current line 639 ⟩;
  fCurrentLine ← fCurrentLine + ´␣´; fPhrase ← Copy(fCurrentLine, 1, length(fCurrentLine));
  fPhrasePos.Col ← 1; fPos.Col ← 0; exit
  end
```
This code is used in section 636.

641. Scanning a comment will effectively replace the start of the comment ("`::`") up to and including the end of the line, with a single space. This will process the comment (§642).

⟨ Skip comments 641 ⟩ ≡
```
k ← Pos(´::´, fCurrentLine);   { Comment }
if (k ≠ 0) then
  begin ProcessComment(fPhrasePos.Line, k, copy(fCurrentLine, k, length(fCurrentLine)));
  delete(fCurrentLine, k + 1, length(fCurrentLine)); fCurrentLine[k] ← ´␣´;
  end
```
This code is used in section 636.

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

⟨ Implementation for scanner.pas 590 ⟩ +≡
procedure *MScannObj.ProcessComment*(*fLine*, *fStart* : *integer*; *cmt* : *string*);
  begin end;

643. Testing if the scanner has exhausted the input stream amounts to checking the current line has been completely read *and* the current source file has arrived at an *t*exttteof state.

⟨ Implementation for scanner.pas 590 ⟩ +≡
function *MScannObj.EndOfText*: *boolean*;
  begin *EndOfText* ← (*fPhrasePos.Col* ≥ *length*(*fCurrentLine*)) ∧ *eof*(*fSourceFile*);
  end;

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

⟨ Implementation for scanner.pas 590 ⟩ +≡
function *MScannObj.IsIdentifierLetter*(*ch* : *char*): *boolean*;
  begin *IsIdentifierLetter* ← *Allowed*[*ch*] ≠ 0;
  end;

645. Constructor. The only way to construct a scanner. This expects an article to be read in *aFileName* and a dictionary to be loaded (*aDctFileName*, loaded with §596). The buffer size for reading *aFileName* is initially #4000.

⟨ Implementation for scanner.pas 590 ⟩ +≡
constructor *MScannObj.InitScanning*(**const** *aFileName*, *aDctFileName*: *string*);
  begin *inherited Init*; *Allowed* ← *DefaultAllowed*; *fTokens.LoadDct*(*aDctFileName*);
  assign(*fSourceFile*, *aFileName*); *fSourceBuffSize* ← #4000; *getmem*(*fSourceBuff*, *fSourceBuffSize*);
  settextbuf(*fSourceFile*, *fSourceBuff*↑, #4000); *reset*(*fSourceFile*); *fCurrentLine* ← ´␣´; *GetToken*;
  end;

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

⟨ Implementation for scanner.pas 590 ⟩ +≡
destructor *MScannObj.Done*;
  begin *close*(*fSourceFile*); *FreeMem*(*fSourceBuff*, *fSourceBuffSize*); *fCurrentLine* ← ´´; *inherited Done*;
  end;

File 17

Format

647. The first step towards disambiguating the meaning of identifiers is to use "formats". Recall from, e.g., Andrzej Trybulec's "Some Features of the Mizar Language" (ESPRIT Workshop, Torino, 1993; `mizar.uwb.edu.pl/project/t` §3) 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.

⟨ _format.pas 647 ⟩ ≡
  ⟨ GNU License 4 ⟩
unit *_formats*;
  interface
  uses *mobjects*, *scanner*, *dicthan*, *xml_inout*;
    ⟨ Declare classes for `_formats.pas` 649 ⟩
  function *CompareFormats*(*aItem1*, *aItem2* : *Pointer*): *Integer*;
  function *In_Format*(*fInFile* : *XMLInStreamPtr*): *MFormatPtr*;
    ⟨ Global variables (`_formats.pas`) 648 ⟩
  implementation
  uses *errhan*, *xml_dict*, *xml_parser*
      mdebug , *info* **end_mdebug**;
    ⟨ Implementation for `_formats.pas` 650 ⟩
  end .

648.

⟨ Global variables (`_formats.pas`) 648 ⟩ ≡
var *gFormatsColl*: *MFormatsList*; *gPriority*: *BinIntFunc*; *gFormatsBase*: *integer*;

This code is used in section 647.

649. Broadly speaking, there are only 3 types of "formats": prefix formats, infix formats, bracket-like formats. These are viewed as "subclasses" of a base *MFormat* object.

We will want to collect the formats from articles referenced by the environment of an article being verified or parsed. This motivates the *MFormatList* object.

⟨ Declare classes for `_formats.pas` 649 ⟩ ≡
  ⟨ Declare *MFormat* object 651 ⟩;
    { **TODO**: add assertions that nr. of all format arguments is equal to the number of visible args (Visible) of a pattern }
  ⟨ Declare *MPrefixFormat* object 653 ⟩;
  ⟨ Declare *MInfixFormat* object 655 ⟩;
  ⟨ Declare *MBracketFormat* object 657 ⟩;
  ⟨ Declare *MFormatsList* object 665 ⟩;

This code is used in section 647.

650. 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.

⟨ Implementation for `_formats.pas` 650 ⟩ ≡
  ⟨ Constructors for derived format classess 652 ⟩
  ⟨ Compare formats 659 ⟩
  ⟨ Implementation for *MFormatsList* 666 ⟩
  ⟨ Read formats from an XML input stream 677 ⟩
  ⟨ Implement *MFormatObj* 678 ⟩

This code is used in section 647.

651. Format base class. All format instances have a lexeme called its *fSymbol*. Recall that *LexemeRec* (§589) is a normalized token using a single character to describe its kind, and an integer to keep track of it (instead of relying on a raw string).

⟨ Declare *MFormat* object 651 ⟩ ≡
  MFormatPtr = ↑*MFormatObj*;
  MFormatObj = **object** (*MObject*)
    fSymbol: *LexemRec*;
    constructor *Init*(*aKind* : *Char*; *aSymNr* : *integer*);
    procedure *Out_Format*(**var** *fOutFile* : *XMLOutStreamObj*; *aFormNr* : *integer*);
  end

This code is used in section 649.

652. The constructor expects the "kind" of the object and its symbol number.

⟨ Constructors for derived format classess 652 ⟩ ≡
constructor *MFormatObj.Init*(*aKind* : *Char*; *aSymNr* : *integer*);
  begin *fSymbol.Kind* ← *aKind*; *fSymbol.Nr* ← *aSymNr*;
  end;

See also sections 654, 656, and 658.

This code is used in section 650.

653. Prefix format object.

⟨ Declare *MPrefixFormat* object 653 ⟩ ≡
  MPrefixFormatPtr = ↑*MPrefixFormatObj*;
  MPrefixFormatObj = **object** (*MFormatObj*)
    fRightArgsNbr: *byte*;
    constructor *Init*(*aKind* : *Char*; *aSymNr*, *aRArgsNbr* : *integer*);
  end

This code is used in section 649.

654. Prefix formats track how many arguments are to the right of the prefix symbol.

⟨ Constructors for derived format classess 652 ⟩ +≡
constructor *MPrefixFormatObj.Init*(*aKind* : *Char*; *aSymNr*, *aRArgsNbr* : *integer*);
  begin *fSymbol.Kind* ← *aKind*; *fSymbol.Nr* ← *aSymNr*; *fRightArgsNbr* ← *aRArgsNbr*;
  end;

655. Infix format object.

⟨ Declare *MInfixFormat* object 655 ⟩ ≡
  MInfixFormatPtr = ↑*MInfixFormatObj*;
  MInfixFormatObj = **object** (*MPrefixFormatObj*)
    fLeftArgsNbr: *byte*;
    constructor *Init*(*aKind* : *Char*; *aSymNr*, *aLArgsNbr*, *aRArgsNbr* : *integer*);
  end

This code is used in section 649.

656. 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.

⟨ Constructors for derived format classess 652 ⟩ +≡
constructor *MInfixFormatObj.Init*(*aKind* : *Char*; *aSymNr*, *aLArgsNbr*, *aRArgsNbr* : *integer*);
  begin *fSymbol.Kind* ← *aKind*; *fSymbol.Nr* ← *aSymNr*; *fLeftArgsNbr* ← *aLArgsNbr*;
  fRightArgsNbr ← *aRArgsNbr*;
  end;

657. Bracket format object.

⟨ Declare *MBracketFormat* object 657 ⟩ ≡
  MBracketFormatPtr = ↑*MBracketFormatObj*;
  MBracketFormatObj = **object** (*MInfixFormatObj*)
    fRightSymbolNr: *integer*;
    fArgsNbr: *byte*;
    constructor *Init*(*aLSymNr*, *aRSymNr*, *aArgsNbr*, *aLArgsNbr*, *aRArgsNbr* : *integer*);
  end

This code is used in section 649.

658. ⟨ Constructors for derived format classess 652 ⟩ +≡
constructor *MBracketFormatObj.Init*(*aLSymNr*, *aRSymNr*, *aArgsNbr*, *aLArgsNbr*, *aRArgsNbr* : *integer*);
  begin *fSymbol.Kind* ← ´K´; *fSymbol.Nr* ← *aLSymNr*; *fRightSymbolNr* ← *aRSymNr*;
  fArgsNbr ← *aArgsNbr*; *fLeftArgsNbr* ← *aLArgsNbr*; *fRightArgsNbr* ← *aRArgsNbr*;
  end;

659. Ordering format objects. We need a *Compare* ordering function on formats. This is a lexicographic ordering on the (kind, number of right symbols, number of arguments, number of left symbols), more or less.

⟨ Compare formats 659 ⟩ ≡
function *CompareFormats*(*aItem1* , *aItem2* : *Pointer*): *Integer*;
  begin *CompareFormats* ← 1;
  if *MFormatPtr*(*aItem1*)↑.*fSymbol*.*Kind* < *MFormatPtr*(*aItem2*)↑.*fSymbol*.*Kind* **then**
    CompareFormats ← −1
  else if *MFormatPtr*(*aItem1*)↑.*fSymbol*.*Kind* = *MFormatPtr*(*aItem2*)↑.*fSymbol*.*Kind* **then**
      ⟨ Compare symbols of the same kind 660 ⟩;
  end;

This code is used in section 650.

660. We then check the indexing number of the symbol. When they are the same, we look at the next "entry" in the tuple.

⟨ Compare symbols of the same kind 660 ⟩ ≡
  if *MFormatPtr*(*aItem1*)↑.*fSymbol*.*Nr* < *MFormatPtr*(*aItem2*)↑.*fSymbol*.*Nr* **then**
    CompareFormats ← −1
  else if *MFormatPtr*(*aItem1*)↑.*fSymbol*.*Nr* = *MFormatPtr*(*aItem2*)↑.*fSymbol*.*Nr* **then**
      ⟨ Compare same kinded symbols with the same number 661 ⟩

This code is used in section 659.

661. 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 symbols.

Functors (`'O'`) and predicates (`'R'`) can be compared as infix symbols.

Left functor brackets (`'K'`) can be compared first with bracket-specific characteristics, then as infix symbols.

⟨ Compare same kinded symbols with the same number 661 ⟩ ≡
  case *MFormatPtr*(*aItem1*)↑.*fSymbol*.*Kind* **of**
  ´J´, ´U´: *CompareFormats* ← 0;
  ´G´, ´L´, ´M´, ´V´: ⟨ Compare prefix symbols 662 ⟩;
  ´O´, ´R´: ⟨ Compare infix symbols 664 ⟩;
  ´K´: ⟨ Compare bracket symbols 663 ⟩;
  endcases

This code is used in section 660.

662. Comparing prefixing symbols, at this points, can only compare the number of arguments to the right.

⟨ Compare prefix symbols 662 ⟩ ≡
  if *MPrefixFormatPtr*(*aItem1*)↑.*fRightArgsNbr* < *MPrefixFormatPtr*(*aItem2*)↑.*fRightArgsNbr* **then**
    CompareFormats ← −1
  else if *MPrefixFormatPtr*(*aItem1*)↑.*fRightArgsNbr* = *MPrefixFormatPtr*(*aItem2*)↑.*fRightArgsNbr* **then**
      CompareFormats ← 0

This code is used in section 661.

663. Comparing bracket symbols first tries to compare the number of symbols to its right. If these are equal, then we try to compare the number of arguments. If these are equal, then we compare them "as if" they were infixing symbols.

⟨ Compare bracket symbols 663 ⟩ ≡
  if *MBracketFormatPtr*(*aItem1*)↑.*fRightSymbolNr* < *MBracketFormatPtr*(*aItem2*)↑.*fRightSymbolNr*
      then *CompareFormats* ← −1
  else if *MBracketFormatPtr*(*aItem1*)↑.*fRightSymbolNr* = *MBracketFormatPtr*(*aItem2*)↑.*fRightSymbolNr*
      then
    if *MBracketFormatPtr*(*aItem1*)↑.*fArgsNbr* < *MBracketFormatPtr*(*aItem2*)↑.*fArgsNbr* **then**
      CompareFormats ← −1
    else if *MBracketFormatPtr*(*aItem1*)↑.*fArgsNbr* = *MBracketFormatPtr*(*aItem2*)↑.*fArgsNbr* **then**
      ⟨ Compare infix symbols 664 ⟩

This code is used in section 661.

664. Comparing infixing symbols compares the number of arguments to the left. If these are equal, then we try to compare the number of arguments to the right. If these are equal, then we return 0.

⟨ Compare infix symbols 664 ⟩ ≡
  if *MInfixFormatPtr*(*aItem1*)↑.*fLeftArgsNbr* < *MInfixFormatPtr*(*aItem2*)↑.*fLeftArgsNbr* **then**
    CompareFormats ← −1
  else if *MInfixFormatPtr*(*aItem1*)↑.*fLeftArgsNbr* = *MInfixFormatPtr*(*aItem2*)↑.*fLeftArgsNbr* **then**
    if *MInfixFormatPtr*(*aItem1*)↑.*fRightArgsNbr* < *MInfixFormatPtr*(*aItem2*)↑.*fRightArgsNbr* **then**
      CompareFormats ← −1
    else if *MInfixFormatPtr*(*aItem1*)↑.*fRightArgsNbr* = *MInfixFormatPtr*(*aItem2*)↑.*fRightArgsNbr*
        then *CompareFormats* ← 0

This code is used in sections 661 and 663.

Section 17.1. LIST OF FORMATS

665. We have a collection of format objects managed by a *MFormatsList* object. There are two groups of public functions: "Lookup" functions (to find the format matching certain parameters), and "Collect" functions (to insert a new format).

⟨ Declare *MFormatsList* object 665 ⟩ ≡
  MFormatsListPtr = ↑*MFormatsList*;
  MFormatsList = **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*): *integer*;
    function *CollectPredForm*(*aSymNr*, *aLArgsNbr*, *aRArgsNbr* : *integer*): *integer*;
  end

This code is used in section 649.

666. We prefix format objects specified by its kind, its symbol number, and the number of arguments it expects.

When the format object is not found, then 0 will be returned. This is a standard convention in these functions to indicate the thing is missing.

⟨ Implementation for *MFormatsList* 666 ⟩ ≡
const *PrefixFormatChars* = [´M´, ´V´, ´U´, ´J´, ´L´, ´G´];

function *MFormatsList.LookUp_PrefixFormat*(*aKind* : *char*; *aSymNr*, *aArgsNbr* : *integer*): *integer*;
  var *lFormat*: *MPrefixFormatObj*; *i*: *integer*;
  begin *MizAssert*(3300, *aKind* ∈ *PrefixFormatChars*);
  lFormat.*Init*(*aKind*, *aSymNr*, *aArgsNbr*);
  if *Find*(@*lFormat*, *i*) **then** *LookUp_PrefixFormat* ← *fIndex*↑[*i*] + 1
  else *LookUp_PrefixFormat* ← 0;
  end;

This code is used in section 650.

667. Looking up an infix functor format (§655). This returns the *index* for the entry.

The contract here is rather confusing. What *should* occur is: if there is a functor symbol with the given left and right number of arguments, then return the index for the entry. Otherwise (when there is no functor symbol) return -1.

What happens instead is these values are incremented, so if the functor symbol with the given number of left and right arguments is contained in position k, then $k+1$ will be returned. If there is no such functor symbol, then 0 will be returned.

function *MFormatsList.LookUp_FuncFormat*(*aSymNr*, *aLArgsNbr*, *aRArgsNbr* : *integer*): *integer*;
  var *lFormat*: *MInfixFormatObj*; *i*: *integer*;
  begin *lFormat.Init*(`'O'`, *aSymNr*, *aLArgsNbr*, *aRArgsNbr*);
  if *Find*(@*lFormat*, *i*) **then** *LookUp_FuncFormat* ← *fIndex*↑[*i*] + 1
  else *LookUp_FuncFormat* ← 0;
  end;

668. Looking up a bracket.

function *MFormatsList.LookUp_BracketFormat*(*aLSymNr*, *aRSymNr*, *aArgsNbr*, *aLArgsNbr*,
    aRArgsNbr : *integer*): *integer*;
  var *lFormat*: *MBracketFormatObj*; *i*: *integer*;
  begin *lFormat.Init*(*aLSymNr*, *aRSymNr*, *aArgsNbr*, *aLArgsNbr*, *aRArgsNbr*);
  if *Find*(@*lFormat*, *i*) **then** *LookUp_BracketFormat* ← *fIndex*↑[*i*] + 1
  else *LookUp_BracketFormat* ← 0;
  end;

669. Looking up a predicate.

function *MFormatsList.LookUp_PredFormat*(*aSymNr*, *aLArgsNbr*, *aRArgsNbr* : *integer*): *integer*;
  var *lFormat*: *MInfixFormatObj*; *i*: *integer*;
  begin *lFormat.Init*(`'R'`, *aSymNr*, *aLArgsNbr*, *aRArgsNbr*);
  if *Find*(@*lFormat*, *i*) **then** *LookUp_PredFormat* ← *fIndex*↑[*i*] + 1
  else *LookUp_PredFormat* ← 0;
  end;

670. Insert a format, if it's missing.

function *MFormatsList.CollectFormat*(*aFormat* : *MFormatPtr*): *integer*;
  var *lFormatNr*, *i*: *integer*;
  begin *lFormatNr* ← 0;
  if ¬*Find*(*aFormat*, *i*) **then**
    begin *lFormatNr* ← *Count* + 1; *Insert*(*aFormat*);
    end;
  CollectFormat ← *lFormatNr*;
  end;

671. Inserting a bracket, if it is missing. Returns the format number for the format, whether it is missing or not.

function *MFormatsList.CollectBracketForm*(*aLSymNr*, *aRSymNr*, *aArgsNbr*, *aLArgsNbr*, *aRArgsNbr* : *integer*): *integer*;
  var *lFormatNr*: *integer*;
  begin *lFormatNr* ← *LookUp_BracketFormat*(*aLSymNr*, *aRSymNr*, *aArgsNbr*, *aLArgsNbr*, *aRArgsNbr*);
  if *lFormatNr* = 0 **then**
    begin *lFormatNr* ← *Count* + 1;
    Insert(*new*(*MBracketFormatPtr*, *Init*(*aLSymNr*, *aRSymNr*, *aArgsNbr*, *aLArgsNbr*, *aRArgsNbr*)));
    end;
  CollectBracketForm ← *lFormatNr*;
  end;

672. 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* ← *LookUp_FuncFormat*(*aSymNr*, *aLArgsNbr*, *aRArgsNbr*);
  if *lFormatNr* = 0 **then**
    begin *lFormatNr* ← *Count* + 1;
    Insert(*new*(*MInfixFormatPtr*, *Init*(´O´, *aSymNr*, *aLArgsNbr*, *aRArgsNbr*)));
    end;
  CollectFuncForm ← *lFormatNr*;
  end;

673. Insert a prefix format if it is missing. Then return the format number for the prefix format, missing or not.

function *MFormatsList.CollectPrefixForm*(*aKind* : *char*; *aSymNr*, *aArgsNbr* : *integer*): *integer*;
  var *lFormatNr*: *integer*;
  begin *lFormatNr* ← *LookUp_PrefixFormat*(*aKind*, *aSymNr*, *aArgsNbr*);
  if *lFormatNr* = 0 **then**
    begin *lFormatNr* ← *Count* + 1; *Insert*(*new*(*MPrefixFormatPtr*, *Init*(*aKind*, *aSymNr*, *aArgsNbr*)));
    end;
  CollectPrefixForm ← *lFormatNr*;
  end;

674. Insert a predicate format, if it is missing. Then return the format number, whether the predicate format is missing or not.

function *MFormatsList.CollectPredForm*(*aSymNr*, *aLArgsNbr*, *aRArgsNbr* : *integer*): *integer*;
  var *lFormatNr*: *integer*;
  begin *lFormatNr* ← *LookUp_PredFormat*(*aSymNr*, *aLArgsNbr*, *aRArgsNbr*);
  if *lFormatNr* = 0 **then**
    begin *lFormatNr* ← *Count* + 1;
    Insert(*new*(*MInfixFormatPtr*, *Init*(´R´, *aSymNr*, *aLArgsNbr*, *aRArgsNbr*)));
    end;
  CollectPredForm ← *lFormatNr*;
  end;

675. Constructor. Construct the empty list of formats.

constructor *MFormatsList.Init*(*ALimit* : *Integer*);
  begin *InitSorted*(*ALimit*, *CompareFormats*);
  end;

676. Constructor. Parse an XML file for formats, and populate a format list object with the file's contents.

constructor *MFormatsList.LoadFormats*(*fName* : *string*);
  var *lEnvFile*: *XMLInStreamPtr*; *lValue*: *integer*; *lLex*: *LexemRec*;
  begin *InitSorted*(100, *CompareFormats*); *lEnvFile* ← *new*(*XMLInStreamPtr*, *OpenFile*(*fName*));
  with *lEnvFile*↑ **do**
    begin *NextElementState*; *XMLASSERT*(*nElName* = *XMLElemName*[*elFormats*]);
    NextElementState;
    while ¬(*nState* = *eEnd*) ∧ (*nElName* = *XMLElemName*[*elFormat*]) **do** *Insert*(*In_Format*(*lEnvFile*));
    gPriority.Init(10);
    while ¬(*nState* = *eEnd*) **do**
      begin *XMLASSERT*(*nElName* = *XMLElemName*[*elPriority*]);
      lLex.Kind ← *GetAttr*(*XMLAttrName*[*atKind*])[1];
      lLex.Nr ← *GetIntAttr*(*XMLAttrName*[*atSymbolNr*]); *MizAssert*(3300, *lLex.Kind* ∈ [´O´, ´L´, ´K´]);
      lValue ← *GetIntAttr*(*XMLAttrName*[*atValue*]); *gPriority.Assign*(*ord*(*lLex.Kind*), *lLex.Nr*, *lValue*);
      AcceptEndState; *NextElementState*;
      end;
    end;
  dispose(*lEnvFile*, *Done*);
  end;

677. We can read exactly one format from an XML input stream.

⟨Read formats from an XML input stream 677⟩ ≡
function *In_Format*(*fInFile* : *XMLInStreamPtr*): *MFormatPtr*;
  var *lLex*: *LexemRec*; *lArgsNbr*, *lLeftArgsNbr*, *lRightSymNr*: *integer*;
  begin with *fInFile*↑ **do**
    begin *lLex.Kind* ← *GetAttr*(*XMLAttrName*[*atKind*])[1];
    lLex.Nr ← *GetIntAttr*(*XMLAttrName*[*atSymbolNr*]);
    lArgsNbr ← *GetIntAttr*(*XMLAttrName*[*atArgNr*]);
    case *lLex.Kind* **of**
    ´O´, ´R´: **begin** *lLeftArgsNbr* ← *GetIntAttr*(*XMLAttrName*[*atLeftArgNr*]);
      In_Format ← *new*(*MInfixFormatPtr*, *Init*(*lLex.Kind*, *lLex.Nr*, *lLeftArgsNbr*,
          lArgsNbr − *lLeftArgsNbr*));
      end;
    ´J´, ´U´, ´V´, ´G´, ´L´, ´M´: *In_Format* ← *new*(*MPrefixFormatPtr*, *Init*(*lLex.Kind*, *lLex.Nr*, *lArgsNbr*));
    ´K´: **begin** *lRightSymNr* ← *GetIntAttr*(*XMLAttrName*[*atRightSymbolNr*]);
      In_Format ← *new*(*MBracketFormatPtr*, *Init*(*lLex.Nr*, *lRightSymNr*, *lArgsNbr*, 0, 0));
      end;
    othercases *RunTimeError*(2019);
    endcases;
    AcceptEndState; *NextElementState*;
    end;
  end;

This code is used in section 650.

678. Conversely, we can print to an output stream an XML representation for a format object.

⟨ Implement *MFormatObj* 678 ⟩ ≡

```
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[atArgNr], MPrefixFormatPtr(@Self)↑.fRightArgsNbr);
    ´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[atArgNr], fArgsNbr);
        Out_XIntAttr(XMLAttrName[atRightSymbolNr], fRightSymbolNr);
        end;
    othercases RuntimeError(3300);
    endcases;
    Out_XElEnd0;
    end;
  end;
```

This code is used in section 650.

679. Given a list of formats, we can store them to an XML file using the previous function.

```
procedure MFormatsList.StoreFormats(fName : string);
  var lEnvFile: XMLOutStreamObj; z: integer;
  begin lEnvFile.OpenFile(fName);
  with lEnvFile do
    begin Out_XElStart0(XMLElemName[elFormats]);
    for z ← 0 to Count − 1 do MFormatPtr(Items↑[z])↑.Out_Format(lEnvFile, z + 1);
    with gPriority do
      for z ← 0 to fCount − 1 do
        begin Out_XElStart(XMLElemName[elPriority]);
        Out_XAttr(XMLAttrName[atKind], chr(fList↑[z].X1));
        Out_XIntAttr(XMLAttrName[atSymbolNr], fList↑[z].X2);
        Out_XIntAttr(XMLAttrName[atValue], fList↑[z].Y); Out_XElEnd0;
        end;
    Out_XElEnd(XMLElemName[elFormats]);
    end;
  lEnvFile.Done;
  end;
```

680. We clean up the formats collection and the priority. The *gPriority* is initialized and populated in other functions. The *gFormatsColl* is used heavily in `parseraddition.pas` and a few other places.

```
procedure DisposeFormats;
  begin gFormatsColl.Done; gPriority.Done;
  end;
```

File 18

Syntax

681. This describes the syntax for the Mizar language, using expressions, subexpressions, blocks, and "items" (statements).

We will need to recall *StackedObj* from `mobjects.pas` (§307).

⟨ syntax.pas 681 ⟩ ≡
  ⟨ GNU License 4 ⟩
unit *syntax*;
  interface
  uses *mobjects*, *errhan*; ⟨ Interface for `syntax.pas` 688 ⟩
  implementation
  uses *mconsole*
    mdebug , *info* **end_mdebug**;
    ⟨ Implementation for `syntax.pas` 683 ⟩
  end .

682. The maximum number of "visible" arguments to an expression is set here, at 10.

⟨ Public constants for `syntax.pas` 682 ⟩ ≡
const *MaxVisArgNbr* = 10;

This code is used in section 688.

683. The implementation for the abstract syntax of Mizar is rather uninteresting, since most of the methods are abstract.

⟨ Implementation for `syntax.pas` 683 ⟩ ≡
  ⟨ Subexpression constructor 714 ⟩
  ⟨ Subexpression destructor 715 ⟩
  ⟨ Expression constructor 711 ⟩
  ⟨ Subexpression procedures 718 ⟩
  ⟨ Create a subexpression for an expression 712 ⟩
  ⟨ Item object implementation 704 ⟩
  ⟨ Block object implementation 694 ⟩
  ⟨ Public procedures implementation for `syntax.pas` 684 ⟩

This code is used in section 681.

684. Destructor wrappers. We have a few public-facing procedures to free the global subexpression, expression, etc., variables describing the state of the parser.

⟨ Public procedures implementation for `syntax.pas` 684 ⟩ ≡
procedure *KillSubexpression*;
  begin if *gSubexpPtr* = **nil then** *RunTimeError*(2144)
  else *dispose*(*gSubexpPtr*, *Done*);
  end;

See also sections 685, 686, and 687.

This code is used in section 683.

685.

⟨ Public procedures implementation for `syntax.pas` 684 ⟩ +≡

procedure *KillExpression*;
  begin if *gExpPtr* = **nil then** *RunTimeError*(2143)
  else *dispose*(*gExpPtr*, *Done*);
  end;

686. This method will not be used until we get to the parser, sadly. I am not sure why there are calls to *DisplayLine* in *KillItem* and *KillBlock*, though.

The *KillItem* is called in exactly two places: (1) *Semicolon* in `parser.pas`, (2) *SchemeBlock*, also in the parser. (And *KillBlock* is called only in the parser, as well.)

⟨ Public procedures implementation for `syntax.pas` 684 ⟩ +≡

procedure *KillItem*;
  begin if *gItemPtr* = **nil then** *RunTimeError*(2142)
  else begin *gItemPtr*↑.*Pop*; *dispose*(*gItemPtr*, *Done*); **end**;
  DisplayLine(*CurPos*.*Line*, *ErrorNbr*);
  end;

687.

⟨ Public procedures implementation for `syntax.pas` 684 ⟩ +≡

procedure *KillBlock*;
  begin if *gBlockPtr* = **nil then** *RunTimeError*(2141)
  else begin *gBlockPtr*↑.*Pop*; *dispose*(*gBlockPtr*, *Done*);
    end;
  DisplayLine(*CurPos*.*Line*, *ErrorNbr*);
  end;

688.

⟨ Interface for `syntax.pas` 688 ⟩ ≡
  ⟨ Public constants for `syntax.pas` 682 ⟩
type ⟨ BlockKinds (`syntax.pas`) 692 ⟩
  ⟨ ItemKinds (`syntax.pas`) 702 ⟩
  ⟨ ExpKinds (`syntax.pas`) 709 ⟩
  ⟨ Block object interface 693 ⟩;
  ⟨ Class declaration for Item object 703 ⟩;
  ⟨ Subexpression object class 713 ⟩;
  ⟨ Expression class declaration 710 ⟩;
  ⟨ Public procedures for `syntax.pas` 689 ⟩
  ⟨ Public variables for `syntax.pas` 690 ⟩

This code is used in section 681.

689. ⟨ Public procedures for `syntax.pas` 689 ⟩ ≡

procedure *KillBlock*;
procedure *KillItem*;
procedure *KillExpression*;
procedure *KillSubexpression*;

This code is used in section 688.

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

⟨ Public variables for `syntax.pas` 690 ⟩ ≡
var *gBlockPtr*: *BlockPtr* = **nil**; *gItemPtr*: *ItemPtr* = **nil**; *gExpPtr*: *ExpressionPtr* = **nil**;
gSubexpPtr: *SubexpPtr* = **nil**;

This code is used in section 688.

Section 18.1. BLOCK OBJECT

691. The Mizar language is block-structured, so we have a Block represent a sequence of statements contained within a block.

This is extended in `parseraddition.pas`.

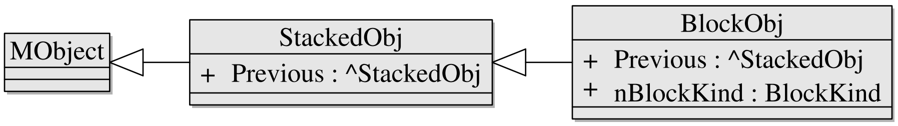

Fig. 1. UML class diagram for Block object class.

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

⟨ BlockKinds (`syntax.pas`) 692 ⟩ ≡
BlockKind = (*blMain*, *blDiffuse*, *blHereby*, *blProof*, *blDefinition*, *blNotation*, *blRegistration*, *blCase*, *blSuppose*, *blPublicScheme*);

This code is used in section 688.

693. ⟨ Block object interface 693 ⟩ ≡
BlockPtr = ↑*BlockObj*;
ItemPtr = ↑*ItemObj*;

BlockObj = **object** (*StackedObj*)
  nBlockKind: *BlockKind*;
  constructor *Init*(*fBlockKind* : *BlockKind*);
  procedure *Pop*; *virtual*; { inheritance }
  destructor *Done*; *virtual*;
  procedure *StartProperText*; *virtual*;
  procedure *ProcessLink*; *virtual*;
  procedure *ProcessRedefine*; *virtual*;
  procedure *ProcessBegin*; *virtual*;
  procedure *ProcessPragma*; *virtual*;
  procedure *StartAtSignProof*; *virtual*;
  procedure *FinishAtSignProof*; *virtual*;
  procedure *FinishDefinition*; *virtual*;
  procedure *CreateItem*(*fItemKind* : *ItemKind*); *virtual*;
  procedure *CreateBlock*(*fBlockKind* : *BlockKind*); *virtual*;
  procedure *StartSchemeDemonstration*; *virtual*;
  procedure *FinishSchemeDemonstration*; *virtual*;
end

This code is used in section 688.

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

⟨ Block object implementation 694 ⟩ ≡
constructor *BlockObj.Init*(*fBlockKind* : *BlockKind*);
  begin *nBlockKind* ← *fBlockKind*; *Previous* ← *gBlockPtr*;
  end;

See also sections 695, 696, 697, 698, 699, and 700.

This code is used in section 683.

695. Note that popping a block object is left for subclasses to handle.

⟨ Block object implementation 694 ⟩ +≡
procedure *BlockObj.Pop*;
  begin end;

696. ⟨ Block object implementation 694 ⟩ +≡
destructor *BlockObj.Done*;
  begin *gBlockPtr* ← *BlockPtr*(*Previous*);
  end;

697. Abstract methods.

⟨ Block object implementation 694 ⟩ +≡
procedure *BlockObj.StartProperText*;
  begin end;
procedure *BlockObj.ProcessRedefine*;
  begin end;
procedure *BlockObj.ProcessLink*;
  begin end;
procedure *BlockObj.ProcessBegin*;
  begin end;
procedure *BlockObj.ProcessPragma*;
  begin end;
procedure *BlockObj.StartAtSignProof*;
  begin end;
procedure *BlockObj.FinishAtSignProof*;
  begin end;
procedure *BlockObj.FinishDefinition*;
  begin end;

698. ⟨ Block object implementation 694 ⟩ +≡
procedure *BlockObj.CreateItem*(*fItemKind* : *ItemKind*);
  begin *gItemPtr* ← *new*(*ItemPtr*, *Init*(*fItemKind*));
  end;

699. ⟨ Block object implementation 694 ⟩ +≡
procedure *BlockObj.CreateBlock*(*fBlockKind* : *BlockKind*);
  begin *gBlockPtr* ← *new*(*BlockPtr*, *Init*(*fBlockKind*));
  end;

700. More abstract methods.

⟨ Block object implementation 694 ⟩ +≡
procedure *BlockObj.StartSchemeDemonstration*;
  begin end;
procedure *BlockObj.FinishSchemeDemonstration*;
  begin end;

Section 18.2. ITEM OBJECTS

701. The class declaration for an *Item* object is depressingly long, with most of its virtual methods not used. The class diagram is worth drawing out.

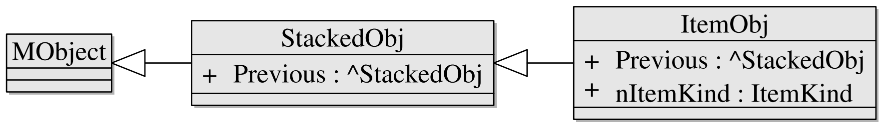

Fig. 2. UML class diagram for Item object class.

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

⟨ ItemKinds (`syntax.pas`) 702 ⟩ ≡

ItemKind = (*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*);

This code is used in section 688.

703. ⟨ Class declaration for Item object 703 ⟩ ≡

ItemObj = **object** (*StackedObj*)
  nItemKind: *ItemKind*;
  constructor *Init*(*fItemKind* : *ItemKind*);
  procedure *Pop*; *virtual*;
  destructor *Done*; *virtual*;
  ⟨ Method declarations for Item object 707 ⟩
end

This code is used in section 688.

704. It is particularly important to note, when constructing an *Item* object, the previous item will *automatically* be set to point to the global *gItem* variable.

⟨ Item object implementation 704 ⟩ ≡

constructor *ItemObj.Init*(*fItemKind* : *ItemKind*);
  begin *nItemKind* ← *fItemKind*; *Previous* ← *gItemPtr*;
  end;
procedure *ItemObj.Pop*;
  begin *DisplayLine*(*CurPos.Line*, *ErrorNbr*);
  end;
destructor *ItemObj.Done*;
  begin *DisplayLine*(*CurPos.Line*, *ErrorNbr*); *gItemPtr* ← *ItemPtr*(*Previous*);
  end;

See also sections 705 and 708.

This code is used in section 683.

705. Creating an expression in an item is handled with this method.

⟨ Item object implementation 704 ⟩ +≡
procedure *ItemObj.CreateExpression*(*fExpKind* : *ExpKind*);
  begin *gExpPtr* ← *new*(*ExpressionPtr*, *Init*(*fExpKind*));
  end;

706. Abstract methods. The methods of the *Item* class can be partitioned into two groups: those which will be implemented by a subclass, and those which will remain "empty" (i.e., whose body is just **begin end**).

⟨ Methods overriden by extended Item class 706 ⟩ ≡
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 *FinishFixedSegment*; *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*;

procedure *ProcessFuncSynonym*; *virtual*;
procedure *ProcessModeSynonym*; *virtual*;

procedure *StartVisible*; *virtual*;
procedure *ProcessVisible*; *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 *ProcessSchemeVariable*; *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*;

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

See also section 1224.

This code is used in sections 707 and 1225.

707. ⟨ Method declarations for Item object 707 ⟩ ≡
⟨ Methods overriden by extended Item class 706 ⟩
procedure *FinishClusterType*; *virtual*;
procedure *FinishSentence*; *virtual*;
procedure *FinishReconsidering*; *virtual*;
procedure *StartNewType*; *virtual*;
procedure *StartCondition*; *virtual*;
procedure *FinishChoice*; *virtual*;
procedure *FinishAssumption*; *virtual*;
procedure *StartEquals*; *virtual*;
procedure *StartOtherwise*; *virtual*;
procedure *StartSpecification*; *virtual*;
procedure *ProcessAttributePattern*; *virtual*;
procedure *StartDefPredicate*; *virtual*;
procedure *CompletePredAntonymByAttr*; *virtual*;
procedure *CompletePredSynonymByAttr*; *virtual*;
procedure *StartPredIdentify*; *virtual*;
procedure *ProcessPredIdentify*; *virtual*;
procedure *CompleteAttrIdentify*; *virtual*;
procedure *StartAttrIdentify*; *virtual*;
procedure *ProcessAttrIdentify*; *virtual*;
procedure *CompletePredIdentify*; *virtual*;
procedure *FinishFuncReduction*; *virtual*;
procedure *StartSethoodProperties*; *virtual*;
procedure *ProcessModePattern*; *virtual*;
procedure *StartPrefix*; *virtual*;
procedure *FinishVisible*; *virtual*;
procedure *FinishSchemeHeading*; *virtual*;
procedure *FinishSchemeDeclaration*; *virtual*;
procedure *StartSchemePremise*; *virtual*;
procedure *StartTheoremBody*; *virtual*;
procedure *FinishTheoremBody*; *virtual*;
procedure *FinishTheorem*; *virtual*;
procedure *FinishReservation*; *virtual*;
procedure *ProcessIterativeStep*; *virtual*;
{ *Justification* }
procedure *StartSchemeReference*; *virtual*;
procedure *StartReferences*; *virtual*;
procedure *ProcessSch*; *virtual*;
procedure *FinishTheLibraryReferences*; *virtual*;
procedure *FinishSchLibraryReferences*; *virtual*;
procedure *FinishReferences*; *virtual*;
procedure *FinishSchemeReference*; *virtual*;
procedure *FinishJustification*; *virtual*;

This code is used in section 703.

708. ⟨Item object implementation 704⟩ +≡

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;
procedure *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;
procedure *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*;

  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;
procedure *ItemObj.FinishGuard*;
  begin end;
procedure *ItemObj.StartOtherwise*;
  begin end;
procedure *ItemObj.ProcessEquals*;
  begin end;
procedure *ItemObj.StartEquals*;
  begin end;
procedure *ItemObj.ProcessCorrectness*;
  begin end;
procedure *ItemObj.FinishSpecification*;
  begin end;
procedure *ItemObj.FinishConstructionType*;
  begin end;
procedure *ItemObj.StartSpecification*;
  begin end;
procedure *ItemObj.StartExpansion*;
  begin end;
procedure *ItemObj.StartConstructionType*;
  begin end;
procedure *ItemObj.StartPredicatePattern*;
  begin end;
procedure *ItemObj.ProcessPredicateSymbol*;
  begin end;
procedure *ItemObj.FinishPredicatePattern*;
  begin end;
procedure *ItemObj.StartFunctorPattern*;
  begin end;
procedure *ItemObj.ProcessFunctorSymbol*;
  begin end;
procedure *ItemObj.FinishFunctorPattern*;

```
  begin end;
procedure ItemObj.ProcessAttrAntonym;
  begin end;
procedure ItemObj.ProcessAttrSynonym;
  begin end;
procedure ItemObj.ProcessPredAntonym;
  begin end;
procedure ItemObj.ProcessPredSynonym;
  begin end;
procedure ItemObj.ProcessFuncSynonym;
  begin end;
procedure ItemObj.CompletePredSynonymByAttr;
  begin end;
procedure ItemObj.CompletePredAntonymByAttr;
  begin end;
procedure ItemObj.ProcessModeSynonym;
  begin end;
procedure ItemObj.StartFuncIdentify;
  begin end;
procedure ItemObj.ProcessFuncIdentify;
  begin end;
procedure 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;
procedure ItemObj.StartSethoodProperties;
  begin end;
procedure ItemObj.FinishSethoodProperties;
  begin end;
procedure ItemObj.StartModePattern;
  begin end;
procedure ItemObj.ProcessModePattern;
```

```
  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;
procedure 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;
procedure ItemObj.ProcessSchemeName;
  begin end;
procedure ItemObj.StartSchemeSegment;
  begin end;
procedure ItemObj.ProcessSchemeVariable;
  begin end;
procedure ItemObj.StartSchemeQualification;
  begin end;
procedure ItemObj.FinishSchemeQualification;
  begin end;
procedure ItemObj.FinishSchemeSegment;
  begin end;
procedure ItemObj.FinishSchemeHeading;
  begin end;
procedure ItemObj.FinishSchemeDeclaration;
  begin end;
procedure ItemObj.FinishSchemeThesis;
  begin end;
procedure ItemObj.StartSchemePremise;
```

begin end;
procedure *ItemObj.FinishSchemePremise*;
begin end;
procedure *ItemObj.StartTheoremBody*;
begin end;
procedure *ItemObj.FinishTheoremBody*;
begin end;
procedure *ItemObj.FinishTheorem*;
begin end;
procedure *ItemObj.StartReservationSegment*;
begin end;
procedure *ItemObj.ProcessReservedIdentifier*;
begin end;
procedure *ItemObj.FinishReservationSegment*;
begin end;
procedure *ItemObj.FinishReservation*;
begin end;
procedure *ItemObj.StartPrivateDefiniendum*;
begin end;
procedure *ItemObj.FinishLocusType*;
begin end;
procedure *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*;

```
  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;
procedure ItemObj.ProcessIterativeStep;
  begin end;
procedure ItemObj.FinishIterativeStep;
  begin end;
```

Section 18.3. EXPRESSIONS

709.

⟨ ExpKinds (`syntax.pas`) 709 ⟩ ≡
  $ExpKind = (exNull, exType, exTerm, exFormula, exResType, exAdjectiveCluster)$;

This code is used in section 688.

710. ⟨ Expression class declaration 710 ⟩ ≡
  $ExpressionPtr = \uparrow ExpressionObj$;
  $ExpressionObj =$ **object** $(MObject)$
    $nExpKind$: $ExpKind$;
    constructor $Init(fExpKind : ExpKind)$;
    procedure $CreateSubexpression$; $virtual$;
  end

This code is used in section 688.

711. Constructor.

⟨ Expression constructor 711 ⟩ ≡
constructor $ExpressionObj.Init(fExpKind : ExpKind)$;
  begin $nExpKind \leftarrow fExpKind$;
  end;

This code is used in section 683.

712. Observe that creating a subexpression (1) allocates a new $SubexpPtr$ on the heap, and (2) mutates the $gSubexpPtr$ global variable.

⟨ Create a subexpression for an expression 712 ⟩ ≡
procedure $ExpressionObj.CreateSubexpression$;
  begin $gSubexpPtr \leftarrow new(SubexpPtr, Init)$;
  end;

This code is used in section 683.

Section 18.4. SUBEXPRESSIONS

713.

⟨ Subexpression object class 713 ⟩ ≡
  SubexpPtr = ↑*SubexpObj*;
  SubexpObj = **object** (*StackedObj*)
    constructor *Init*;
    destructor *Done*; *virtual*;
    ⟨ Empty method declarations for *SubexpObj* 717 ⟩
  end

This code is used in section 688.

714. Constructor. Importantly, constructing a new *Subexp* object will initialize its *Previous* field to point to the global *gSubexpPtr* object.

⟨ Subexpression constructor 714 ⟩ ≡
constructor *SubexpObj.Init*;
  begin *Previous* ← *gSubexpPtr*;
  end;

This code is used in section 683.

715. Destructor.

⟨ Subexpression destructor 715 ⟩ ≡
destructor *SubexpObj.Done*;
  begin *gSubexpPtr* ← *SubexpPtr*(*Previous*);
  end;

This code is used in section 683.

716. The remaining methods for subexpression objects are empty.

⟨ Methods implemented by subclasses of *SubexpObj* 716 ⟩ ≡
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 *FinishType*; *virtual*;
procedure *CompleteType*; *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*;

{ 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 mozliwosci 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 *CompleteAttributeArguments*; *virtual*; { + }
procedure *FinishAttributeArguments*; *virtual*; { + }
procedure *CompleteAdjectiveCluster*; *virtual*; { + }
procedure *CompleteClusterTerm*; *virtual*;

{ *Errors Recovery* }
procedure *InsertIncorrTerm*; *virtual*;
procedure *InsertIncorrType*; *virtual*;
procedure *InsertIncorrBasic*; *virtual*;
procedure *InsertIncorrFormula*; *virtual*;

See also section 1387.

This code is used in sections 717 and 1388.

717. ⟨ Empty method declarations for *SubexpObj* 717 ⟩ ≡
⟨ Methods implemented by subclasses of *SubexpObj* 716 ⟩

procedure *FinishSample*; *virtual*;
procedure *ProcessThe*; *virtual*;
procedure *StartArgument*; *virtual*;
procedure *ProcessLeftParenthesis*; *virtual*;
procedure *ProcessRightParenthesis*; *virtual*;
procedure *StartAtomicFormula*; *virtual*;
procedure *ProcessHolds*; *virtual*;
procedure *FinishQuantified*; *virtual*;
procedure *ProcessNot*; *virtual*;
procedure *ProcessDoesNot*; *virtual*;
procedure *StartAdjectiveCluster*; *virtual*;
procedure *FinishAdjectiveCluster*; *virtual*;
procedure *FinishAttributes*; *virtual*;
procedure *CompleteAttributes*; *virtual*;
procedure *CompleteClusterType*; *virtual*;
procedure *FinishEquality*; *virtual*;

This code is used in section 713.

718.

⟨ Subexpression procedures 718 ⟩ ≡
procedure *SubexpObj.StartAttributes*;
  begin end;
procedure *SubexpObj.StartAdjectiveCluster*;
  begin end;
procedure *SubexpObj.FinishAdjectiveCluster*;
  begin end;
procedure *SubexpObj.ProcessNon*;
  begin end;
procedure *SubexpObj.ProcessAttribute*;
  begin end;
procedure *SubexpObj.FinishAttributes*;
  begin end;
procedure *SubexpObj.CompleteAttributes*;
  begin end;
procedure *SubexpObj.StartAttributeArguments*;
  begin end;
procedure *SubexpObj.CompleteAttributeArguments*;
  begin end;
procedure *SubexpObj.FinishAttributeArguments*;
  begin end;
procedure *SubexpObj.CompleteAdjectiveCluster*;
  begin end;
procedure *SubexpObj.CompleteClusterTerm*;
  begin end;
procedure *SubexpObj.CompleteClusterType*;
  begin end;
procedure *SubexpObj.ProcessSimpleTerm*;
  begin end;
procedure *SubexpObj.ProcessQua*;
  begin end;
procedure *SubexpObj.FinishQualifiedTerm*;
  begin end;
procedure *SubexpObj.ProcessExactly*;
  begin end;
procedure *SubexpObj.StartArgument*;
  begin end;
procedure *SubexpObj.FinishArgument*;
  begin end;
procedure *SubexpObj.FinishTerm*;
  begin end;
procedure *SubexpObj.StartType*;
  begin end;
procedure *SubexpObj.ProcessModeSymbol*;
  begin end;
procedure *SubexpObj.FinishType*;
  begin end;
procedure *SubexpObj.CompleteType*;
  begin end;
procedure *SubexpObj.StartLongTerm*;
  begin end;

procedure *SubexpObj.FinishLongTerm*;
  begin end;
procedure *SubexpObj.FinishArgList*;
  begin end;
procedure *SubexpObj.ProcessFunctorSymbol*;
  begin end;
procedure *SubexpObj.StartFraenkelTerm*;
  begin end;
procedure *SubexpObj.FinishSample*;
  begin end;
procedure *SubexpObj.StartPostqualification*;
  begin end;
procedure *SubexpObj.StartPostqualificationSpecyfication*;
  begin end;
procedure *SubexpObj.StartPostqualifyingSegment*;
  begin end;
procedure *SubexpObj.ProcessPostqualifiedVariable*;
  begin end;
procedure *SubexpObj.FinishPostqualifyingSegment*;
  begin end;
procedure *SubexpObj.FinishFraenkelTerm*;
  begin end;
procedure *SubexpObj.StartSimpleFraenkelTerm*;
  begin end;
procedure *SubexpObj.FinishSimpleFraenkelTerm*;
  begin end;
procedure *SubexpObj.StartPrivateTerm*;
  begin end;
procedure *SubexpObj.FinishPrivateTerm*;
  begin end;
procedure *SubexpObj.StartBracketedTerm*;
  begin end;
procedure *SubexpObj.FinishBracketedTerm*;
  begin end;
procedure *SubexpObj.StartAggregateTerm*;
  begin end;
procedure *SubexpObj.FinishAggregateTerm*;
  begin end;
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;
procedure *SubexpObj.StartAtomicFormula*;
  begin end;
procedure *SubexpObj.ProcessAtomicFormula*;
  begin end;
procedure *SubexpObj.ProcessPredicateSymbol*;
  begin end;
procedure *SubexpObj.ProcessRightSideOfPredicateSymbol*;
  begin end;
procedure *SubexpObj.FinishPredicativeFormula*;
  begin end;
procedure *SubexpObj.FinishRightSideOfPredicativeFormula*;
  begin end;
procedure *SubexpObj.StartMultiPredicativeFormula*;
  begin end;
procedure *SubexpObj.FinishMultiPredicativeFormula*;
  begin end;
procedure *SubexpObj.FinishQualifyingFormula*;
  begin end;
procedure *SubexpObj.FinishAttributiveFormula*;
  begin end;
procedure *SubexpObj.StartPrivateFormula*;
  begin end;
procedure *SubexpObj.FinishPrivateFormula*;
  begin end;
procedure *SubexpObj.ProcessContradiction*;
  begin end;
procedure *SubexpObj.ProcessNot*;
  begin end;
procedure *SubexpObj.ProcessDoesNot*;
  begin end;
procedure *SubexpObj.ProcessNegative*;
  begin end;
procedure *SubexpObj.ProcessNegation*;
  begin end;
procedure *SubexpObj.StartRestriction*;
  begin end;
procedure *SubexpObj.FinishRestriction*;
  begin end;
procedure *SubexpObj.ProcessHolds*;
  begin end;
procedure *SubexpObj.ProcessBinaryConnective*;
  begin end;
procedure *SubexpObj.FinishBinaryFormula*;
  begin end;

procedure *SubexpObj.ProcessFlexDisjunction*;
  begin end;
procedure *SubexpObj.ProcessFlexConjunction*;
  begin end;
procedure *SubexpObj.FinishFlexDisjunction*;
  begin end;
procedure *SubexpObj.FinishFlexConjunction*;
  begin end;
procedure *SubexpObj.StartQualifiedSegment*;
  begin end;
procedure *SubexpObj.StartQualifyingType*;
  begin end;
procedure *SubexpObj.FinishQualifiedSegment*;
  begin end;
procedure *SubexpObj.FinishQuantified*;
  begin end;
procedure *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 683.

File 19

MScanner

719. We have the MScanner module transform an article (an input file) into a stream of tokens.

⟨ scanner.pas 588 ⟩ +≡
  ⟨ GNU License 4 ⟩
unit *mscanner*;
  interface
  uses *errhan*, *mobjects*, *scanner*;
    ⟨ Public interface for MScanner 720 ⟩
  implementation
  uses *mizenv*;
    ⟨ Implementation for MScanner 726 ⟩;

  end .

720. Public types. We have enumerated types for each construction we'll encounter in Mizar.

⟨ Public interface for MScanner 720 ⟩ ≡
type ⟨ Token kinds for MScanner 724 ⟩;
  CorrectnessKind = (*syCorrectness*, *syCoherence*, *syCompatibility*, *syConsistency*, *syExistence*, *syUniqueness*, *syReducibility*);

  PropertyKind = (*sErrProperty*, *sySymmetry*, *syReflexivity*, *syIrreflexivity*, *syAssociativity*, *syTransitivity*, *syCommutativity*, *syConnectedness*, *syAsymmetry*, *syIdempotence*, *syInvolutiveness*, *syProjectivity*, *sySethood*, *syAbstractness*);

  LibraryReferenceKind = (*syThe*, *syDef*, *sySch*);

  DirectiveKind = (*syVocabularies*, *syNotations*, *syDefinitions*, *syTheorems*, *sySchemes*, *syRegistrations*, *syConstructors*, *syRequirements*, *syEqualities*, *syExpansions*);

  ⟨ Token type for MScanner 721 ⟩;

See also sections 722 and 723.

This code is used in section 719.

721. Token type for MScanner.

⟨ Token type for MScanner 721 ⟩ ≡
  Token = **record** *Kind*: *TokenKind*;
    Nr: *integer*;
    Spelling: *string*;
  end

This code is used in section 720.

722. Constants for MScanner

⟨ Public interface for MScanner 720 ⟩ +≡
const { Homonymic and special symbols in buildin vocabulery }
  { Homonymic Selector Symbol }
  StrictSym = 1; { "strict" }
  { Homonymic Mode Symbol }
  SetSym = 1; { 'set' }
  { Homonymic Predicate Symbol }
  EqualitySym = 1; { '=' }
  { Homonymic Circumfix Symbols }
  SquareBracket = 1; { '[' ']' }
  CurlyBracket = 2; { "–" """ }
  RoundedBracket = 3; { "(" ")" }
  scTooLongLineErrorNr = 200; { Error number: Too long line }
  ⟨ Token names for MScanner 725 ⟩;
CorrectnessName: **array** [*CorrectnessKind*] **of** *string* = (´correctness´, ´coherence´, ´compatibility´, ´consistency´, ´existence´, ´uniqueness´, ´reducibility´);
PropertyName: **array** [*PropertyKind*] **of** *string* = (´´, ´symmetry´, ´reflexivity´, ´irreflexivity´, ´associativity´, ´transitivity´, ´commutativity´, ´connectedness´, ´asymmetry´, ´idempotence´, ´involutiveness´, ´projectivity´, ´sethood´, ´abstractness´);
LibraryReferenceName: **array** [*LibraryReferenceKind*] **of** *string* = (´the´, ´def´, ´sch´);
DirectiveName: **array** [*DirectiveKind*] **of**
  string = (´vocabularies´, ´notations´, ´definitions´, ´theorems´, ´schemes´, ´registrations´, ´constructors´, ´requirements´, ´equalities´, ´expansions´);
PlaceHolderName: **array** [1 . . 10] **of**
  string = (´$1´, ´$2´, ´$3´, ´$4´, ´$5´, ´$6´, ´$7´, ´$8´, ´$9´, ´$10´);
  Unexpected = *sErrProperty*;

723. Public facing procedures and global variables. Of particular importance, the global variable *gScanner* is declared here.

⟨ Public interface for MScanner 720 ⟩ +≡
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*;

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

⟨ Token kinds for MScanner 724 ⟩ ≡

```
TokenKind = (syT0,  { #0 }
  syT1,  { #1 }
  syT2,  { #2 }
  syT3,  { #3 }
  syT4,  { #4 }
  syT5,  { #5 }
  syT6,  { #6 }
  syT7,  { #7 }
  syT8,  { #8 }
  syT9,  { #9 }
  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 }
  syT31,  { #31 }
  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 }
  sy_notation,  { - #45 }
  sy_Ellipsis,  { .  #46 }
  sy_proof,  { / #47 }
  syT48,  { 0 #48 }
  syT49,  { 1 #49 }
```

```
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, { O #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 }
```

```
syT102, { f #102 }
syT103, { g #103 }
sy_with, { h #104 }
syT105, { i #105 }
syT106, { j #106 }
syT107, { k #107 }
syT108, { l #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, { #135 }
syT136, { #136 }
syT137, { #137 }
syT138, { #138 }
syT139, { #139 }
sy_if = 140, { #140 }
syT141, { #141 }
syT142, { #142 }
syT143, { #143 }
sy_is = 144, { #144 }
sy_are, { #145 }
syT146, { #146 }
sy_otherwise, { #147 }
syT148, { #148 }
syT149, { #149 }
syT150, { #150 }
syT151, { #151 }
syT152, { #152 }
syT153, { #153 }
```

$syT154$, { #154 }
$sy\_ex = 156$, { #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, { #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, { #236 }
sy_struct, { #237 }
sy_exactly, { #238 }
sy_mode, { #239 }
sy_iff, { #240 }
sy_func, { #241 }
sy_pred, { #242 }
sy_implies, { #243 }
sy_st, { #244 }
sy_holds, { #245 }
sy_provided, { #246 }
sy_means, { #247 }
sy_of, { #248 }
sy_defpred, { #249 }
sy_deffunc, { #250 }
sy_such, { #251 }
sy_that, { #252 }
sy_aggregate, { #253 }
sy_and { #254 });

This code is used in section 720.

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

⟨ Token names for MScanner 725 ⟩ ≡

```
TokenName: array [TokenKind] of string = ('',  {#0 }
    '',  { #1 }
    '',  { #2 }
    '',  { #3 }
    '',  { #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 }
```

```
'', {2 #50 }
'', {3 #51 }
'', {4 #52 }
'', {5 #53 }
'', {6 #54 }
'', {7 #55 }
'', {8 #56 }
'', {9 #57 }
':', {: #58 }
';', {; #59 }
'now', {< #60 }
'=', {= #61 }
'end', {> #62 }
'', {? #63 }
'', {@ #64 }
'', {A #65 }
'', {B #66 }
'', {C #67 }
'vocabularies', {D #68 }
'', {E #69 }
'', {F #70 }
'', {G #71 }
'', {H #72 }
'', {I #73 }
'', {J #74 }
'', {K #75 }
'', {L #76 }
'', {M #77 }
'', {N #78 }
'', {O #79 }
'', {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}
```

```
'', { f #102 }
'', { g #103 }
'with', { h #104 }
'', { i #105 }
'', { j #106 }
'', { k #107 }
'', { l #108 }
'', { m #109 }
'', { n #110 }
'', { o #111 }
'', { p #112 }
'', { q #113 }
'wrt', { r #114 }
'', { s #115 }
'to', { t #116 }
'', { u #117 }
'', { v #118 }
'when', { w #119 }
'axiom', { x #120 }
'', { y #121 }
'', { z #122 }
'{', {  #123 }
'', { | #124 }
'}', {  #125 }
'', { ~ #126 }
'T127', { #127 }
'', { #128 }
'T129', { #129 }
'', { #130 }
'T131', { #131 }
'', { #132 }
'', { #133 }
'', { #134 }
'correctness', { #135 }
'T136', { #136 }
'', { #137 }
'', { #138 }
'', { #139 }
'if', { #140 }
'', { #141 }
'', { #142 }
'', { #143 }
'is', { #144 }
'are', { #145 }
'', { #146 }
'otherwise', { #147 }
'', { #148 }
'', { #149 }
'', { #150 }
'', { #151 }
'T152', { #152 }
'', { #153 }
```

```
´´, {#154}
´´, {#155}
´ex´, {#156}
´for´, {#157}
´´, {#158}
´define´, {#159}
´´, {#160}
´being´, {#161}
´over´, {#162}
´´, {#163}
´canceled´, {#164}
´do´, {#165}
´does´, {#166}
´or´, {#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}
´.=´, {#193}
´´, {#194}
´´, {#195}
´synonym´, {#196}
´antonym´, {#197}
´´, {#198}
´´, {#199}
´let´, {#200}
´take´, {#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}
'', {#217}
'', {#218}
'', {#219}
'', {#220}
'', {#221}
'', {#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}
'func', {#241}
'pred', {#242}
'implies', {#243}
'st', {#244}
'holds', {#245}
'provided', {#246}
'means', {#247}
'of', {#248}
'defpred', {#249}
'deffunc', {#250}
'such', {#251}
'that', {#252}
'aggregate', {#253}
'and' {#254})
```

This code is used in section 722.

726. Reading a token. This tokenizes a Mizar article, using the scanner's *GetToken* method. We can trace this *GetToken* back to its implementation (§629). This, in turn, depends on the *SliceIt* method (§612).

This method is used to determine the next token in `parser.pas`'s *Parse* function.

This assumes that *StartScanner* (§729) has been invoked already, which initializes the *CurWord* token and other variables.

Also important to observe: the *Kind* of the token is populated here.

⟨Implementation for MScanner 726⟩ ≡
procedure *ReadToken*;
  begin *PrevWord* ← *CurWord*; *PrevPos* ← *CurPos*; *CurWord* ← *AheadWord*; *CurPos* ← *AheadPos*;
    { '_' is not allowed in an identifiers in the text proper }
  if (*CurWord*.*Kind* = *sy_Begin*) **then** *gScanner*↑.*Allowed*[´_´] ← 0;
  if (*CurWord*.*Kind* = *sy_Error*) ∧ (*CurWord*.*Nr* = *scTooLongLineErrorNr*) **then**
    ErrImm(*CurWord*.*Nr*);
  gScanner↑.*GetToken*;
  AheadWord.*Kind* ← *TokenKind*(*gScanner*↑.*fLexem*.*Kind*); *AheadWord*.*Nr* ← *gScanner*↑.*fLexem*.*Nr*;
  AheadWord.*Spelling* ← *gScanner*↑.*fStr*; *AheadPos* ← *gScanner*↑.*fPos*;
  end;

See also sections 727, 728, 729, 730, 731, 732, and 733.

This code is used in section 719.

727. Loading a proof file. The `.prf` file is a file containing numerals, and its usage eludes me. The format consists of multiple lines:

Line 1: Three non-negative integers are on the first line "M S P"
Line 2: Contains M non-negative integers separated by a single whitespace
Line 3: Contains S non-negative integers separated by a single whitespace
Line 4: Contains P non-negative integers separated by a single whitespace.

This function loads the contents of the `.prf` file. This initializes the global variables *ModeMaxArgs*, *StructureModeMaxArgs*, *PredMaxArgs*, then populates them.

⟨Implementation for MScanner 726⟩ +≡
procedure *LoadPrf*(**const** *aPrfFileName*: *string*);
  var *lPrf*: *text*; *lModeMaxArgsSize*, *lStructModeMaxArgsSize*, *lPredMaxArgsSize*, *i*, *lInt*, *r*: *integer*;
  begin *assign*(*lPrf*, *aPrfFileName* + ´.prf´); *reset*(*lPrf*);
  Read(*lPrf*, *lModeMaxArgsSize*, *lStructModeMaxArgsSize*, *lPredMaxArgsSize*);
  ModeMaxArgs.*Init*(*lModeMaxArgsSize* + 1); *r* ← *ModeMaxArgs*.*Insert*(0);
  StructModeMaxArgs.*Init*(*lStructModeMaxArgsSize* + 1); *r* ← *StructModeMaxArgs*.*Insert*(0);
  PredMaxArgs.*Init*(*lPredMaxArgsSize* + 1); *r* ← *PredMaxArgs*.*Insert*(0);
  for *i* ← 1 **to** *lModeMaxArgsSize* **do**
    begin *Read*(*lPrf*, *lInt*); *r* ← *ModeMaxArgs*.*Insert*(*lInt*);
    end;
  for *i* ← 1 **to** *lStructModeMaxArgsSize* **do**
    begin *Read*(*lPrf*, *lInt*); *r* ← *StructModeMaxArgs*.*Insert*(*lInt*);
    end;
  for *i* ← 1 **to** *lPredMaxArgsSize* **do**
    begin *Read*(*lPrf*, *lInt*); *r* ← *PredMaxArgs*.*Insert*(*lInt*);
    end;
  close(*lPrf*);
  end;

728. We cleanup after using the `.prf` file.

⟨ Implementation for MScanner 726 ⟩ +≡
procedure *DisposePrf* ;
  begin *ModeMaxArgs*.*Done*; *PredMaxArgs*.*Done*; *StructModeMaxArgs*.*Done*;
  end;

729. We construct an MScann object to scan a file.

⟨ Implementation for MScanner 726 ⟩ +≡
procedure *StartScaner*;
  begin *CurPos*.*Line* ← 1; *CurPos*.*Col* ← 0; *AheadWord*.*Kind* ← *TokenKind*(*gScanner*↑.*fLexem*.*Kind*);
  AheadWord.*Nr* ← *gScanner*↑.*fLexem*.*Nr*; *AheadWord*.*Spelling* ← *gScanner*↑.*fStr*;
  AheadPos ← *gScanner*↑.*fPos*;
  end;

730. We initialize a scanner for a file.

⟨ Implementation for MScanner 726 ⟩ +≡
procedure *InitSourceFile*(**const** *aFileName*, *aDctFileName*: *string*);
  begin *new*(*gScanner*, *InitScanning*(*aFileName*, *aDctFileName*)); *StartScaner*;
  end;

731. When we're done with a scanner, we call the destructor for the MScanner.

⟨ Implementation for MScanner 726 ⟩ +≡
procedure *CloseSourceFile*;
  begin *dispose*(*gScanner*, *Done*);
  end;

732. We can combine the previous functions together to initialize a scanner for a file (an article) and its dictionary file.

⟨ Implementation for MScanner 726 ⟩ +≡
procedure *InitScanning*(**const** *aFileName*, *aDctFileName*: *string*);
  begin *gScanner* ← *new*(*MScannPtr*, *InitScanning*(*aFileName*, *aDctFileName*)); *StartScaner*;
  LoadPrf (*aDctFileName*);
  end;

733. We cleanup after scanning, saving a dictionary XML file to an "`.idx`" file. This uses the global variable *EnvFileName* declared in `mizenv.pas` (§24).

⟨ Implementation for MScanner 726 ⟩ +≡
procedure *FinishScanning*;
  begin *gScanner*↑.*fIdents*.*SaveXDct*(*EnvFileName* + ´`.idx`´); *CloseSourceFile*; *DisposePrf* ;
  end;

File 20

Abstract Syntax

734. A crucial step in any interpreter, compiler, or proof assistant is to transform the concrete syntax into an abstract syntax tree. This module provides all the classes for the abstract syntax tree *of expressions, types, and formulas* in Mizar. The abstract syntax tree for "statements" will be found in the "Weakly Strict Text Proper" module.

This is a bit, well, "Java-esque", in the sense that each different kind of node in the abstract syntax tree is represented by a different class. If you don't know abstract syntax trees, I can heartily recommend Bob Nystrom's *Crafting Interpreters* (Ch. 5: Representing Code) for an overview.

I'll be quoting from the grammar for Mizar as we go along, since the class hierarchy names their classes after the nonterminal symbols in the grammar. (It's what anyone would do.) You can find a local copy of the grammar on most UNIX machines with Mizar installed located at `/usr/local/doc/Mizar/syntax.txt`, which you can study at your leisure.

735. Warning: There is a lot of boiler plate code in the constructors and destructors. I am going to pass over them without much comment, because they are monotonous and uninteresting. The more interesting part will be discussed with the class declarations for each kind of node. I will simply entitle the paragraphs "Constructor" to indicate I am recognizing their existence and moving on.

⟨ abstract_syntax.pas 735 ⟩ ≡
  ⟨ GNU License 4 ⟩
unit *abstract_syntax*;
  interface uses *errhan*, *mobjects*, *syntax*;
    ⟨ Interface for abstract syntax 737 ⟩
  implementation
    ⟨ Implementation of abstract syntax 736 ⟩
  end .

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

⟨ Implementation of abstract syntax 736 ⟩ ≡
  ⟨ Variable AST constructor 739 ⟩
  ⟨ Qualified segment AST constructor 742 ⟩
  ⟨ Adjective expression AST constructor 748 ⟩
  ⟨ Adjective AST constructor 752 ⟩
  ⟨ Negated adjective AST constructor 750 ⟩
  ⟨ Implementing term AST 757 ⟩
  ⟨ Implementing type AST 799 ⟩
  ⟨ Implementing formula AST 811 ⟩
  ⟨ Within expression AST implementation 1664 ⟩

This code is used in section 735.

737. The interface consists mostly of classes, as well as a few enumerated types. The gambit resembles what we would do if we were programming in C: define an `enum TermSort`, then introduce a `struct TermAstNode {enum TermSort sort;}` to act as an abstract base class for terms (and do likewise for formulas, types, etc.). This allows us to use "struct inheritance" in C, as Bob Nystrom's *Crafting Interpreters* (Ch. 19) calls it.

⟨ Interface for abstract syntax 737 ⟩ ≡
type ⟨ Abstract base class for types 797 ⟩;
  ⟨ Abstract base class for terms 753 ⟩;
  ⟨ Abstract base class for formulas 808 ⟩;
  ⟨ Adjective expression (abstract syntax tree) 747 ⟩;
  ⟨ Negated adjective expression (abstract syntax tree) 749 ⟩;
  ⟨ Adjective (abstract syntax tree) 751 ⟩;
    { Auxiliary structures }
  ⟨ Variable (abstract syntax tree) 738 ⟩;
  ⟨ Qualified segment (abstract syntax tree) 741 ⟩;
  ⟨ Classes for terms (abstract syntax tree) 755 ⟩
  ⟨ Classes for type (abstract syntax tree) 798 ⟩
  ⟨ Classes for formula (abstract syntax tree) 810 ⟩
    {———————————————————————————-}
  ⟨ Class for Within expression 1663 ⟩;

This code is used in section 735.

738. Variable. A variable in the abstract syntax tree is basically a de Bruijn index, in the sense that it is represented by an integer in the metalanguage (PASCAL).

Logicians may feel uncomfortable at variables being outside the term syntax tree. But what logicians think of as "variables" in first-order logic, Mizar calls them "Simple Terms" (§756).

⟨ Variable (abstract syntax tree) 738 ⟩ ≡
  VariablePtr = ↑*VariableObj*;
  VariableObj = **object** (*MObject*)
    nIdent: *integer*; { identifier number }
    nVarPos: *Position*;
    constructor *Init*(**const** *aPos*: *Position*; *aIdentNr*: *integer*);
  end

This code is used in section 737.

739. Constructor.

⟨ Variable AST constructor 739 ⟩ ≡
constructor *VariableObj.Init*(**const** *aPos*: *Position*; *aIdentNr*: *integer*);
  begin *nIdent* ← *aIdentNr*; *nVarPos* ← *aPos*;
  end;

This code is used in section 736.

740. Qualified segment. A qualified segment refers to situations in, e.g.,"`consider` ⟨*qualified-segment*⟩$^{+}$ `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:

```
Qualified-Variables = Implicitly-Qualified-Variables
  | Explicitly-Qualified-Variables
  | Explicitly-Qualified-Variables "," Implicitly-Qualified-Variables .
Implicitly-Qualified-Variables = Variables .
Explicitly-Qualified-Variables = Qualified-Segment {"," Qualified-Segment }.
Qualified-Segment = Variables Qualification .
Variables = Variable-Identifier {"," Variable-Identifier }.
Qualification = ( "being" | "be" ) Type-Expression .
```

We will implement `Qualified-Variables` as an array of pointers to *QualifiedSegment* objects, each one being either implicit or explicit.

741. Abstract base class for qualified segments. We have *implicitly* qualified segments and *explicitly* qualified segments, which are "both" qualified segments. Object-oriented yoga teaches us to describe this situation using a "qualified segment" abstract base class, and then extend it with two subclasses.

⟨ Qualified segment (abstract syntax tree) 741 ⟩ ≡

```
SegmentKind = (ikImplQualifiedSegm, ikExplQualifiedSegm);
QualifiedSegmentPtr = ↑QualifiedSegmentObj;
QualifiedSegmentObj = object (MObject)
  nSegmPos: Position;
  nSegmentSort: SegmentKind;
  constructor Init(const aPos: Position; aSort: SegmentKind);
end
```

See also sections 743 and 745.

This code is used in section 737.

742. Constructor.

⟨ Qualified segment AST constructor 742 ⟩ ≡

```
constructor QualifiedSegmentObj.Init(const aPos: Position; aSort: SegmentKind);
  begin nSegmPos ← aPos; nSegmentSort ← aSort;
  end;
```

See also sections 744 and 746.

This code is used in section 736.

743. Implicitly qualified segments. When we use "reserved variables" in the qualifying segment, we can suppress the type ascription (i.e., the "`being` ⟨*Type*⟩"). This makes the typing *implicit*. Hence the name *implicitly* qualified segments (the types are implicitly given).

⟨ Qualified segment (abstract syntax tree) 741 ⟩ +≡

```
ImplicitlyQualifiedSegmentPtr = ↑ImplicitlyQualifiedSegmentObj;
ImplicitlyQualifiedSegmentObj = object (QualifiedSegmentObj)
  nIdentifier: VariablePtr;
  constructor Init(const aPos: Position; aIdentifier: VariablePtr);
  destructor Done; virtual;
end ;
```

744. Constructor. The constructors and destructors for implicitly qualified segments are straightforward.

⟨ Qualified segment AST constructor 742 ⟩ +≡
constructor *ImplicitlyQualifiedSegmentObj.Init*(**const** *aPos*: *Position*; *aIdentifier*: *VariablePtr*);
  begin *inherited Init*(*aPos*, *ikImplQualifiedSegm*); *nIdentifier* ← *aIdentifier*;
  end;

destructor *ImplicitlyQualifiedSegmentObj.Done*;
  begin *dispose*(*nIdentifier*, *Done*);
  end;

745. Explicitly qualified segment. The other possibility in Mizar is that we will have "explicitly typed variables" in the qualifying segment. The idea is that, in Mizar, we can permit the following situation:

```
consider x,y,z being set such that ...
```

This means the three variables x, y, z are explicitly qualified variables with the type "`set`". We represent this using one *ExplicitlyQualifiedSegment* object, a vector for the identifiers (x, y, z) and a pointer to their type (`set`).

⟨ Qualified segment (abstract syntax tree) 741 ⟩ +≡
  ExplicitlyQualifiedSegmentPtr = ↑*ExplicitlyQualifiedSegmentObj*;
  ExplicitlyQualifiedSegmentObj = **object** (*QualifiedSegmentObj*)
    nIdentifiers: *PList*; { of identifier numbers }
    nType: *TypePtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aIdentifiers*: *PList*; *aType*: *TypePtr*);
    destructor *Done*; *virtual*;
  end

746. The constructors and destructors for explicitly qualified segments are straightforward.

⟨ Qualified segment AST constructor 742 ⟩ +≡
constructor *ExplicitlyQualifiedSegmentObj.Init*(**const** *aPos*: *Position*;
    aIdentifiers: *PList*;
    aType: *TypePtr*);
  begin *inherited Init*(*aPos*, *ikExplQualifiedSegm*); *nIdentifiers* ← *aIdentifiers*; *nType* ← *aType*;
  end;

destructor *ExplicitlyQualifiedSegmentObj.Done*;
  begin *dispose*(*nIdentifiers*, *Done*); *dispose*(*nType*, *Done*);
  end;

747. Attributes. Attributes can have arguments *preceding* it. The relevant part of the Mizar grammar, I think, is:

```
Adjective-Cluster = { Adjective } .
Adjective = [ "non" ] [ Adjective-Arguments ] Attribute-Symbol .
```

⟨ Adjective expression (abstract syntax tree) 747 ⟩ ≡
  AdjectiveSort = (*wsNegatedAdjective*, *wsAdjective*);

  AdjectiveExpressionPtr = ↑*AdjectiveExpressionObj*;
  AdjectiveExpressionObj = **object** (*MObject*)
    nAdjectivePos: *Position*;
    nAdjectiveSort: *AdjectiveSort*;
    constructor *Init*(**const** *aPos*: *Position*; *aSort*: *AdjectiveSort*);
    destructor *Done*; *virtual*;
  end

This code is used in section 737.

748. ⟨ Adjective expression AST constructor 748 ⟩ ≡

constructor *AdjectiveExpressionObj.Init*(**const** *aPos*: *Position*; *aSort*: *AdjectiveSort*);
  begin *nAdjectivePos* ← *aPos*; *nAdjectiveSort* ← *aSort*;
  end;
destructor *AdjectiveExpressionObj.Done*;
  begin end;

This code is used in section 736.

749. Negated adjective. We represent an adjective using the EBNF grammar (c.f., the WSM article-related function *InWSMizFileObj.Read_Adjective:AdjectiveExpressionPtr*):

```
Negated-Adjective ::= "non" Adjective-Expr;
Positive-Adjective ::= [Adjective-Arguments] Attribute-Symbol;
Adjective-Expr ::= Negated-Adjective | Positive-Adjective;
```

Hence we only really need a pointer to the "adjective being negated".

⟨ Negated adjective expression (abstract syntax tree) 749 ⟩ ≡

  NegatedAdjectivePtr = ↑*NegatedAdjectiveObj*;
  NegatedAdjectiveObj = **object** (*AdjectiveExpressionObj*)
    nArg: *AdjectiveExpressionPtr*; { of TermPtr, visible arguments }
    constructor *Init*(**const** *aPos*: *Position*; *aArg*: *AdjectiveExpressionPtr*);
    destructor *Done*; *virtual*;
  end

This code is used in section 737.

750. Constructor.

⟨ Negated adjective AST constructor 750 ⟩ ≡

constructor *NegatedAdjectiveObj.Init*(**const** *aPos*: *Position*; *aArg*: *AdjectiveExpressionPtr*);
  begin *inherited Init*(*aPos*, *wsNegatedAdjective*); *nArg* ← *aArg*;
  end;
destructor *NegatedAdjectiveObj.Done*;
  begin *dispose*(*nArg*, *Done*);
  end;

This code is used in section 736.

751. Adjective objects. ⟦This is the preferred node for later intermediate representations for attributes, since *nNegated* is a field in the class.⟧

⟨ Adjective (abstract syntax tree) 751 ⟩ ≡

  AdjectivePtr = ↑*AdjectiveObj*;
  AdjectiveObj = **object** (*AdjectiveExpressionObj*)
    nAdjectiveSymbol: *integer*;
    nNegated: *boolean*;
    nArgs: *PList*; { of TermPtr, visible arguments }
    constructor *Init*(**const** *aPos*: *Position*; *aAdjectiveNr*: *integer* ; *aArgs*: *PList*);
    destructor *Done*; *virtual*;
  end

This code is used in section 737.

752. Constructor.

⟨ Adjective AST constructor 752 ⟩ ≡
constructor *AdjectiveObj.Init*(**const** *aPos*: *Position*; *aAdjectiveNr*: *integer*; *aArgs*: *PList*);
  begin *inherited Init*(*aPos*, *wsAdjective*); *nAdjectiveSymbol* ← *aAdjectiveNr*; *nArgs* ← *aArgs*;
  end;
destructor *AdjectiveObj.Done*;
  begin *dispose*(*nArgs*, *Done*);
  end;

This code is used in section 736.

Section 20.1. TERMS (ABSTRACT SYNTAX TREE)

753. We have an abstract base class for terms, along with the "sorts" (syntactic subclasses) allowed. This allows, e.g., formulas, to refer to terms without knowing the sort of term involved. The UML class diagram for term:

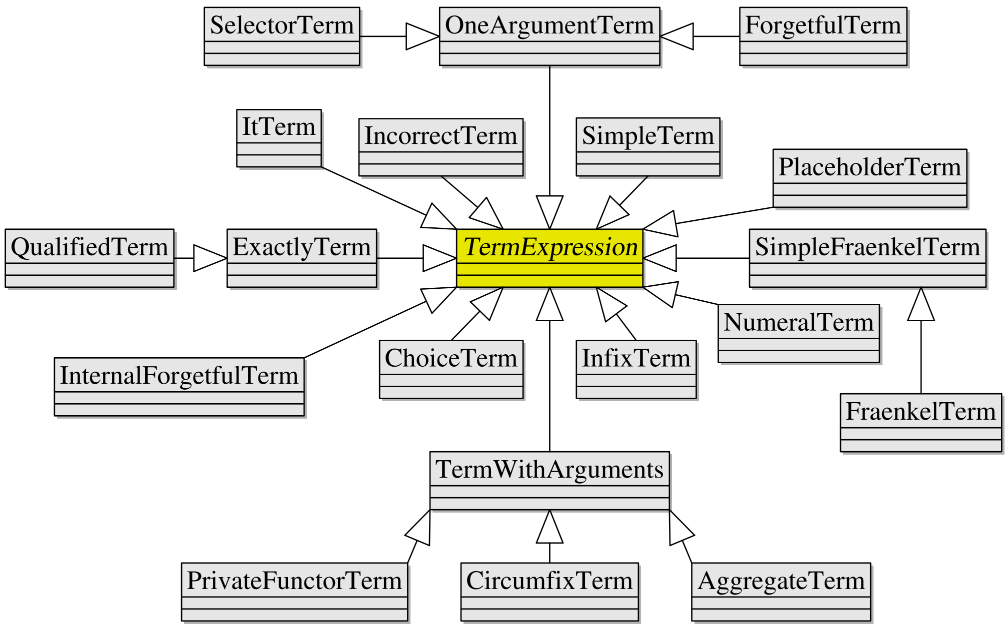

Fig. 3. UML class diagram for abstract syntax tree for terms.

The arrows indicate inheritance, pointing from the subclass to the parent superclass. The abstract base class *TermExpression* is italicized, but it is so difficult to distinguish we have colored it yellow.

NOTE: the class UML diagram may be missing a few descendents of *TermExpression*, but it contains the important subclasses which I could fit into it.

⟨Abstract base class for terms 753⟩ ≡
  TermSort = (*wsErrorTerm*, *wsPlaceholderTerm*, *wsNumeralTerm*, *wsSimpleTerm*,
    wsPrivateFunctorTerm, *wsInfixTerm*, *wsCircumfixTerm*, *wsAggregateTerm*, *wsForgetfulFunctorTerm*,
    wsInternalForgetfulFunctorTerm, *wsSelectorTerm*, *wsInternalSelectorTerm*, *wsQualificationTerm*,
    wsGlobalChoiceTerm, *wsSimpleFraenkelTerm*, *wsFraenkelTerm*, *wsItTerm*, *wsExactlyTerm*);
  TermPtr = ↑*TermExpressionObj*;
  TermExpressionObj = **object** (*MObject*)
    nTermSort: *TermSort*;
    nTermPos: *Position*;
    end

This code is used in section 737.

754. The grammar for term expressions in Mizar as stated in `syntax.txt`:

```
Term-Expression = "(" Term-Expression ")"
  | [ Arguments ] Functor-Symbol [ Arguments ]
  | Left-Functor-Bracket Term-Expression-List Right-Functor-Bracket
  | Functor-Identifier "(" [ Term-Expression-List ] ")"
  | Structure-Symbol "(#" Term-Expression-List "#)"
  | "the" Structure-Symbol "of" Term-Expression
  | Variable-Identifier
  | "{" Term-Expression { Postqualification } ":" Sentence "}"
  | "the" "set" "of" "all" Term-Expression { Postqualification }
  | Numeral
  | Term-Expression "qua" Type-Expression
  | "the" Selector-Symbol "of" Term-Expression
  | "the" Selector-Symbol
  | "the" Type-Expression
  | Private-Definition-Parameter
  | "it" .
```

But I think it might be clearer if we view it using the equivalent grammar:

```
Term-Expression = "(" Term-Expression ")"
  | [ Arguments ] Functor-Symbol [ Arguments ]
  | Left-Functor-Bracket Term-Expression-List Right-Functor-Bracket
  | Functor-Identifier "(" [ Term-Expression-List ] ")"
  | Aggregate-Term
  | Forgetful-Functor-Term
  | Variable-Identifier
  | Fraenkel-Term
  | Numeral
  | Qualified-Term
  | Selector-Functor
  | Internal-Selector-Functor
  | Choice-Term
  | Private-Definition-Parameter
  | "it" .
Aggregate-Term = Structure-Symbol "(#" Term-Expression-List "#)" .
Choice-Term = "the" Type-Expression.
Forgetful-Functor-Term = "the" Structure-Symbol "of" Term-Expression.
Fraenkel-Term = "{" Term-Expression {Postqualification} ":" Sentence "}"
  | "the" "set" "of" "all" Term-Expression { Postqualification }.
Internal-Selector-Functor = "the" Selector-Symbol.
Selector-Functor = "the" Selector-Symbol "of" Term-Expression.
Qualified-Term = Term-Expression "qua" Type-Expression.
```

755. Class structure for this syntax tree.

⟨ Classes for terms (abstract syntax tree) 755 ⟩ ≡
{ Terms }
⟨ Simple term (abstract syntax tree) 756 ⟩;
⟨ Placeholder term (abstract syntax tree) 758 ⟩;
⟨ Numeral term (abstract syntax tree) 760 ⟩;
⟨ Infix term (abstract syntax tree) 762 ⟩;
⟨ Terms with arguments (abstract syntax tree) 764 ⟩;
⟨ Circumfix term (abstract syntax tree) 766 ⟩;
⟨ Private functor term (abstract syntax tree) 768 ⟩;
⟨ One-argument term (abstract syntax tree) 770 ⟩;
⟨ Selector term (abstract syntax tree) 772 ⟩;
⟨ Internal selector term (abstract syntax tree) 774 ⟩;
⟨ Aggregate term (abstract syntax tree) 776 ⟩;
⟨ Forgetful functor (abstract syntax tree) 778 ⟩;
⟨ Internal forgetful functors (abstract syntax tree) 780 ⟩;
⟨ Fraenkel terms (abstract syntax tree) 782 ⟩;
⟨ Exactly term (abstract syntax tree) 788 ⟩;
⟨ Qualified term (abstract syntax tree) 786 ⟩;
⟨ Choice term (abstract syntax tree) 790 ⟩;
⟨ "It" term (abstract syntax tree) 792 ⟩;
⟨ Incorrect term (abstract syntax tree) 794 ⟩;

This code is used in section 737.

756. Simple terms. Mizar describes variables *as terms* as a *SimpleTerm*.

⟨ Simple term (abstract syntax tree) 756 ⟩ ≡
SimpleTermPtr = ↑*SimpleTermObj*;
SimpleTermObj = **object** (*TermExpressionObj*)
nIdent: *integer*; { identifier number }
constructor *Init*(**const** *aPos*: *Position*; *aIdentNr*: *integer*);
end

This code is used in section 755.

757. Constructors.

⟨ Implementing term AST 757 ⟩ ≡
constructor *SimpleTermObj*.*Init*(**const** *aPos*: *Position*; *aIdentNr*: *integer*);
begin *nTermPos* ← *aPos*; *nTermSort* ← *wsSimpleTerm*; *nIdent* ← *aIdentNr*;
end;

See also sections 759, 761, 763, 765, 767, 769, 771, 773, 775, 777, 779, 781, 783, 785, 787, 789, 791, 793, and 795.

This code is used in section 736.

758. Placeholder terms. These are the parameters "`$1`", "`$2`", etc., which appear in a private functor "`deffunc Foo(object) = ...`".

⟨ Placeholder term (abstract syntax tree) 758 ⟩ ≡
PlaceholderTermPtr = ↑*PlaceholderTermObj*; { placeholder }
PlaceholderTermObj = **object** (*TermExpressionObj*)
nLocusNr: *integer*; { $1, ... }
constructor *Init*(**const** *aPos*: *Position*; *aLocusNr*: *integer*);
end

This code is used in section 755.

759. Constructor.

⟨ Implementing term AST 757 ⟩ +≡
constructor *PlaceholderTermObj.Init*(**const** *aPos*: *Position*; *aLocusNr*: *integer*);
  begin *nTermPos* ← *aPos*; *nTermSort* ← *wsPlaceholderTerm*; *nLocusNr* ← *aLocusNr*;
  end;

760. Numeral terms. Mizar can handle 32-bit integers. If we wanted to extend this to, say, arbitrary precision arithmetic, then we would want to modify this class (and a few other places).

⟨ Numeral term (abstract syntax tree) 760 ⟩ ≡
  NumeralTermPtr = ↑*NumeralTermObj*;
  NumeralTermObj = **object** (*TermExpressionObj*)
    nValue: *integer*;
    constructor *Init*(**const** *aPos*: *Position*; *aValue*: *integer*);
  end

This code is used in section 755.

761. Constructor.

⟨ Implementing term AST 757 ⟩ +≡
constructor *NumeralTermObj.Init*(**const** *aPos*: *Position*; *aValue*: *integer*);
  begin *nTermPos* ← *aPos*; *nTermSort* ← *wsNumeralTerm*; *nValue* ← *aValue*;
  end;

762. Infix terms. When we have infix binary operators, they are terms with arguments on both sides of it. For example $x + 2$ will have "+" be an infix term with arguments $(x, 2)$.

We *could* permit multiple arguments on the left-hand side (and on the right-hand side), but they are comma-separated in Mizar. This could happen in finite group theory, for example, "`p -signalizer_over H,G`" has two arguments on the right but only one argument on the left.

⟨ Infix term (abstract syntax tree) 762 ⟩ ≡
  InfixTermPtr = ↑*InfixTermObj*;
  InfixTermObj = **object** (*TermExpressionObj*)
    nFunctorSymbol: *integer*;
    nLeftArgs, *nRightArgs*: *PList*;
    constructor *Init*(**const** *aPos*: *Position*; *aFunctorNr*: *integer*; *aLeftArgs*, *aRightArgs*: *PList*);
    destructor *Done*; *virtual*;
  end

This code is used in section 755.

763. Constructor.

⟨ Implementing term AST 757 ⟩ +≡
constructor *InfixTermObj.Init*(**const** *aPos*: *Position*;
    aFunctorNr: *integer*;
    aLeftArgs, *aRightArgs*: *PList*);
  begin *nTermPos* ← *aPos*; *nTermSort* ← *wsInfixTerm*; *nFunctorSymbol* ← *aFunctorNr*;
  nLeftArgs ← *aLeftArgs*; *nRightArgs* ← *aRightArgs*;
  end;

destructor *InfixTermObj.Done*;
  begin *dispose*(*nLeftArgs*, *Done*); *dispose*(*nRightArgs*, *Done*);
  end;

764. Terms with arguments. This class seems to be used only internally to the `abstract_syntax.pas` module. Recalling the UML class diagram (§753), we remember there are three sublcasses to this: private functor terms (which appear in Mizar when we use "`deffunc F(...) = ...`"), circumfix ("bracketed") terms, and aggregate terms (when we construct an instance of a structure).

⟨ Terms with arguments (abstract syntax tree) 764 ⟩ ≡
  TermWithArgumentsPtr = ↑*TermWithArgumentsObj*;
  TermWithArgumentsObj = **object** (*TermExpressionObj*)
    nArgs: *PList*;
    constructor *Init*(**const** *aPos*: *Position*; *aKind*: *TermSort*; *aArgs*: *PList*);
    destructor *Done*; *virtual*;
  end

This code is used in section 755.

765. Constructor.

⟨ Implementing term AST 757 ⟩ +≡
constructor *TermWithArgumentsObj.Init*(**const** *aPos*: *Position*; *aKind*: *TermSort*; *aArgs*: *PList*);
  begin *nTermPos* ← *aPos*; *nTermSort* ← *aKind*; *nArgs* ← *aArgs*;
  end;
destructor *TermWithArgumentsObj.Done*;
  begin *dispose*(*nArgs*, *Done*);
  end;

766. Circumfix terms. We can introduce different types of brackets in Mizar. For example, for groups, we have the commutator of group elements `[.x,y.]`. These "bracketed terms" are referred to as circumfix terms.

⟨ Circumfix term (abstract syntax tree) 766 ⟩ ≡
  CircumfixTermPtr = ↑*CircumfixTermObj*;
  CircumfixTermObj = **object** (*TermWithArgumentsObj*)
    nLeftBracketSymbol, *nRightBracketSymbol*: *integer*;
    constructor *Init*(**const** *aPos*: *Position*; *aLeftBracketNr*, *aRightBracketNr*: *integer*; *aArgs*: *PList*);
    destructor *Done*; *virtual*;
  end

This code is used in section 755.

767. Constructor.

⟨ Implementing term AST 757 ⟩ +≡
constructor *CircumfixTermObj.Init*(**const** *aPos*: *Position*;
    aLeftBracketNr, *aRightBracketNr*: *integer*;
    aArgs: *PList*);
  begin *inherited Init*(*aPos*, *wsCircumfixTerm*, *aArgs*); *nLeftBracketSymbol* ← *aLeftBracketNr*;
  nRightBracketSymbol ← *aRightBracketNr*;
  end;
destructor *CircumfixTermObj.Done*;
  begin *dispose*(*nArgs*, *Done*);
  end;

768. Private functor terms. We introduce private functor terms in Mizar when we have "`defpred F(...) = ...`".

⟨ Private functor term (abstract syntax tree) 768 ⟩ ≡
  PrivateFunctorTermPtr = ↑*PrivateFunctorTermObj*;
  PrivateFunctorTermObj = **object** (*TermWithArgumentsObj*)
    nFunctorIdent: *integer*;
    constructor *Init*(**const** *aPos*: *Position*; *aFunctorIdNr*: *integer*; *aArgs*: *PList*);
    destructor *Done*; *virtual*;
  end

This code is used in section 755.

769. Constructor.

⟨ Implementing term AST 757 ⟩ +≡
constructor *PrivateFunctorTermObj.Init*(**const** *aPos*: *Position*; *aFunctorIdNr*: *integer*; *aArgs*: *PList*);
  begin *inherited Init*(*aPos*, *wsPrivateFunctorTerm*, *aArgs*); *nFunctorIdent* ← *aFunctorIdNr*;
  end;
destructor *PrivateFunctorTermObj.Done*;
  begin *dispose*(*nArgs*, *Done*);
  end;

770. One-argument terms. Recalling the UML class diagram for terms (§753), we remember the class for *OneArgument* terms are either selector terms ("`the` ⟨*field*⟩ `of` ⟨*aggregate*⟩") or forgetful functors ("`the` ⟨*structure*⟩ `of` ⟨*aggregate*⟩").

⟨ One-argument term (abstract syntax tree) 770 ⟩ ≡
  OneArgumentTermPtr = ↑*OneArgumentTermObj*;
  OneArgumentTermObj = **object** (*TermExpressionObj*)
    nArg: *TermPtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aKind*: *TermSort*; *aArg*: *TermPtr*);
    destructor *Done*; *virtual*;
  end

This code is used in section 755.

771. Constructor.

⟨ Implementing term AST 757 ⟩ +≡
constructor *OneArgumentTermObj.Init*(**const** *aPos*: *Position*; *aKind*: *TermSort*; *aArg*: *TermPtr*);
  begin *nTermPos* ← *aPos*; *nTermSort* ← *aKind*; *nArg* ← *aArg*;
  end;
destructor *OneArgumentTermObj.Done*;
  begin *dispose*(*nArg*, *Done*);
  end;

772. Selector terms. When we have an aggregate term (i.e., an instance of a structure), we want to refer to fields of the structure. This is done with selector terms. [[The selector number refers to the position in the underlying tuple of the structure instance.]]

⟨ Selector term (abstract syntax tree) 772 ⟩ ≡
  SelectorTermPtr = ↑*SelectorTermObj*;
  SelectorTermObj = **object** (*OneArgumentTermObj*)
    nSelectorSymbol: *integer*;
    constructor *Init*(**const** *aPos*: *Position*; *aSelectorNr*: *integer*; *aArg*: *TermPtr*);
    destructor *Done*; *virtual*;
  end

This code is used in section 755.

773. Constructor.

⟨ Implementing term AST 757 ⟩ +≡
constructor *SelectorTermObj.Init*(**const** *aPos*: *Position*; *aSelectorNr*: *integer*; *aArg*: *TermPtr*);
  begin *inherited Init*(*Apos*, *wsSelectorTerm*, *aArg*); *nSelectorSymbol* ← *aSelectorNr*;
  end;
destructor *SelectorTermObj.Done*;
  begin *dispose*(*nArg*, *Done*);
  end;

774. Internal selector terms. An "internal selector" term refers to the case where we have in Mizar "`the` ⟨*selector*⟩" treated as a term.

⟨ Internal selector term (abstract syntax tree) 774 ⟩ ≡
  InternalSelectorTermPtr = ↑*InternalSelectorTermObj*;
  InternalSelectorTermObj = **object** (*TermExpressionObj*)
    nSelectorSymbol: *integer*;
    constructor *Init*(**const** *aPos*: *Position*; *aSelectorNr*: *integer*);
  end

This code is used in section 755.

775. Constructor.

⟨ Implementing term AST 757 ⟩ +≡
constructor *InternalSelectorTermObj.Init*(**const** *aPos*: *Position*; *aSelectorNr*: *integer*);
  begin *nTermPos* ← *aPos*; *nTermSort* ← *wsInternalSelectorTerm*; *nSelectorSymbol* ← *aSelectorNr*;
  end;

776. Aggregate terms. When we construct a new instance of a structure, well, that's a term. Such terms are called "aggregate terms" in Mizar.

⟨ Aggregate term (abstract syntax tree) 776 ⟩ ≡
  AggregateTermPtr = ↑*AggregateTermObj*;
  AggregateTermObj = **object** (*TermWithArgumentsObj*)
    nStructSymbol: *integer*;
    constructor *Init*(**const** *aPos*: *Position*; *aStructSymbol*: *integer*; *aArgs*: *PList*);
    destructor *Done*; *virtual*;
  end

This code is used in section 755.

777. Constructor.

⟨ Implementing term AST 757 ⟩ +≡
constructor *AggregateTermObj.Init*(**const** *aPos*: *Position*; *aStructSymbol*: *integer*; *aArgs*: *PList*);
  begin *inherited Init*(*aPos*, *wsAggregateTerm*, *aArgs*); *nStructSymbol* ← *aStructSymbol*;
  end;
destructor *AggregateTermObj.Done*;
  begin *dispose*(*nArgs*, *Done*);
  end;

778. Forgetful functors. When we have structure inheritance in Mizar, say structure B extends structure 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).

⟨ Forgetful functor (abstract syntax tree) 778 ⟩ ≡
  ForgetfulFunctorTermPtr = ↑*ForgetfulFunctorTermObj*;
  ForgetfulFunctorTermObj = **object** (*OneArgumentTermObj*)
    nStructSymbol: *integer*;
    constructor *Init*(**const** *aPos*: *Position*; *aStructSymbol*: *integer*; *aArg*: *TermPtr*);
    destructor *Done*; *virtual*;
  end

This code is used in section 755.

779. Constructor.

⟨ Implementing term AST 757 ⟩ +≡
constructor *ForgetfulFunctorTermObj.Init*(**const** *aPos*: *Position*; *aStructSymbol*: *integer*;
      aArg: *TermPtr*);
  begin *inherited Init*(*aPos*, *wsForgetfulFunctorTerm*, *aArg*); *nStructSymbol* ← *aStructSymbol*;
  end;
destructor *ForgetfulFunctorTermObj.Done*;
  begin *dispose*(*nArg*, *Done*);
  end;

780. Internal forgetful functors. When we omit the "structure instance" b in a forgetful functor term — e.g., when we have "`the A`" — then we have an "internal forgetful functor" (named analogous to internal selectors).

⟨ Internal forgetful functors (abstract syntax tree) 780 ⟩ ≡
  InternalForgetfulFunctorTermPtr = ↑*InternalForgetfulFunctorTermObj*;
  InternalForgetfulFunctorTermObj = **object** (*TermExpressionObj*)
    nStructSymbol: *integer*;
    constructor *Init*(**const** *aPos*: *Position*; *aStructSymbol*: *integer*);
  end

This code is used in section 755.

781. Constructor.

⟨ Implementing term AST 757 ⟩ +≡
constructor *InternalForgetfulFunctorTermObj.Init*(**const** *aPos*: *Position*; *aStructSymbol*: *integer*);
  begin *nTermPos* ← *aPos*; *nTermSort* ← *wsInternalForgetfulFunctorTerm*;
  nStructSymbol ← *aStructSymbol*;
  end;

782. Simple Fraenkel terms. Fraenkel terms are set-builder notation in Mizar. But "simple" Fraenkel terms occurs when we have "`the set of all` ⟨*termexpr*⟩".

⟨ Fraenkel terms (abstract syntax tree) 782 ⟩ ≡
  SimpleFraenkelTermPtr = ↑*SimpleFraenkelTermObj*;
  SimpleFraenkelTermObj = **object** (*TermExpressionObj*)
    nPostqualification: *PList*; { of segments }
    nSample: *TermPtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aPostqual*: *PList*; *aSample*: *TermPtr*);
    destructor *Done*; *virtual*;
  end ;

See also section 784.

This code is used in section 755.

783. Constructor.

⟨ Implementing term AST 757 ⟩ +≡
constructor *SimpleFraenkelTermObj.Init*(**const** *aPos*: *Position*; *aPostqual*: *PList*; *aSample*: *TermPtr*);
  begin *nTermPos* ← *aPos*; *nTermSort* ← *wsSimpleFraenkelTerm*; *nPostqualification* ← *aPostqual*;
  nSample ← *aSample*;
  end;
destructor *SimpleFraenkelTermObj.Done*;
  begin *dispose*(*nSample*, *Done*);
  end;

784. Fraenkel terms. Fraenkel terms are sets given by set-builder notation, usually they look like

$$\{f(\vec{t})\ \texttt{where}\ \vec{t}\ \texttt{being}\ \vec{T}\ :\ P[\vec{t}]\}$$

This is technically a higher-order object (look, it takes a functor f and a predicate P).

⟨ Fraenkel terms (abstract syntax tree) 782 ⟩ +≡
  FraenkelTermPtr = ↑*FraenkelTermObj*;
  FraenkelTermObj = **object** (*SimpleFraenkelTermObj*)
    nFormula: *FormulaPtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aPostqual*: *PList*; *aSample*: *TermPtr*; *aFormula*:
        FormulaPtr);
    destructor *Done*; *virtual*;
  end

785. Constructor.

⟨ Implementing term AST 757 ⟩ +≡
constructor *FraenkelTermObj.Init*(**const** *aPos*: *Position*;
    aPostqual: *PList*;
    aSample: *TermPtr*;
    aFormula: *FormulaPtr*);
  begin *nTermPos* ← *aPos*; *nTermSort* ← *wsFraenkelTerm*; *nPostqualification* ← *aPostqual*;
  nSample ← *aSample*; *nFormula* ← *aFormula*;
  end;
destructor *FraenkelTermObj.Done*;
  begin *dispose*(*nSample*, *Done*); *dispose*(*nPostqualification*, *Done*); *dispose*(*nFormula*, *Done*);
  end;

786. Qualified terms. We may wish to explicitly type cast a term (e.g., "`term qua newType`"), which is what Mizar calls a "qualified term".

⟨ Qualified term (abstract syntax tree) 786 ⟩ ≡
  QualifiedTermPtr = ↑*QualifiedTermObj*;
  QualifiedTermObj = **object** (*ExactlyTermObj*)
    nQualification: *TypePtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aSubject*: *TermPtr*; *aType*: *TypePtr*);
    destructor *Done*; *virtual*;
  end

This code is used in section 755.

787. Constructor.

⟨ Implementing term AST 757 ⟩ +≡
constructor *QualifiedTermObj.Init*(**const** *aPos*: *Position*; *aSubject*: *TermPtr*; *aType*: *TypePtr*);
  begin *nTermPos* ← *aPos*; *nTermSort* ← *wsQualificationTerm*; *nSubject* ← *aSubject*;
  nQualification ← *aType*;
  end;
destructor *QualifiedTermObj.Done*;
  begin *dispose*(*nSubject*, *Done*); *dispose*(*nQualification*, *Done*);
  end;

788. Exactly terms. This is the base class for qualified terms. It does not appear to be used anywhere outside the abstract syntax module.

⟨ Exactly term (abstract syntax tree) 788 ⟩ ≡
  ExactlyTermPtr = ↑*ExactlyTermObj*;
  ExactlyTermObj = **object** (*TermExpressionObj*)
    nSubject: *TermPtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aSubject*: *TermPtr*);
    destructor *Done*; *virtual*;
  end

This code is used in section 755.

789. Constructor.

⟨ Implementing term AST 757 ⟩ +≡
constructor *ExactlyTermObj.Init*(**const** *aPos*: *Position*; *aSubject*: *TermPtr*);
  begin *nTermPos* ← *aPos*; *nTermSort* ← *wsExactlyTerm*; *nSubject* ← *aSubject*;
  end;
destructor *ExactlyTermObj.Done*;
  begin *dispose*(*nSubject*, *Done*);
  end;

790. Choice terms. This refers to "`the` ⟨*type*⟩" terms. It is a "global choice term" of sorts, except it "operates" on soft types instead of arbitrary predicates.

⟨ Choice term (abstract syntax tree) 790 ⟩ ≡
  ChoiceTermPtr = ↑*ChoiceTermObj*;
  ChoiceTermObj = **object** (*TermExpressionObj*)
    nChoiceType: *TypePtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aType*: *TypePtr*);
    destructor *Done*; *virtual*;
  end

This code is used in section 755.

791. Constructor.

⟨ Implementing term AST 757 ⟩ +≡
constructor *ChoiceTermObj.Init*(**const** *aPos*: *Position*; *aType*: *TypePtr*);
  begin *nTermPos* ← *aPos*; *nTermSort* ← *wsGlobalChoiceTerm*; *nChoiceType* ← *aType*;
  end;
destructor *ChoiceTermObj.Done*;
  begin *dispose*(*nChoiceType*, *Done*);
  end;

792. It terms. When we define a new mode [type] or functors [terms], Mizar introduces an anaphoric keyword "`it`" referring to an example of the mode (resp., to the term being defined). Here I borrow the scary phrase "anaphoric" from Lisp macros, so blame Paul Graham for this pretentiousness.

⟨ "It" term (abstract syntax tree) 792 ⟩ ≡
  ItTermPtr = ↑*ItTermObj*;
  ItTermObj = **object** (*TermExpressionObj*)
    constructor *Init*(**const** *aPos*: *Position*);
  end

This code is used in section 755.

793. Constructor.

⟨ Implementing term AST 757 ⟩ +≡
constructor *ItTermObj.Init*(**const** *aPos*: *Position*);
  begin *nTermPos* ← *aPos*; *nTermSort* ← *wsItTerm*;
  end;

794. Incorrect terms. Generically, when we run into an error of some kind, we represent the term with an *Incorrect* term instance. This will allow Mizar to continue working when the user goofed.

⟨ Incorrect term (abstract syntax tree) 794 ⟩ ≡
  IncorrectTermPtr = ↑*IncorrectTermObj*;
  IncorrectTermObj = **object** (*TermExpressionObj*)
    constructor *Init*(**const** *aPos*: *Position*);
  end

This code is used in section 755.

795. Constructor.

⟨ Implementing term AST 757 ⟩ +≡
constructor *IncorrectTermObj.Init*(**const** *aPos*: *Position*);
  begin *nTermPos* ← *aPos*; *nTermSort* ← *wsErrorTerm*;
  end;

Section 20.2. TYPES (ABSTRACT SYNTAX TREE)

796. 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 "$\forall x.\,\mathtt{Foo}[x]\land Q[x] \implies P[x]$".

797. We have an abstract base class for types.

⟨ Abstract base class for types 797 ⟩ ≡
  TypeSort = (*wsErrorType*, *wsStandardType*, *wsStructureType*, *wsClusteredType*, *wsReservedDscrType*);
    { Initial structures }
  TypePtr = ↑*TypeExpressionObj*;
  TypeExpressionObj = **object** (*MObject*)
    nTypeSort: *TypeSort*;
    nTypePos: *Position*;
    end

This code is used in section 737.

798. Radix type. A "radix type" refers to any type of the form "⟨*RadixType*⟩ `of` T_1 `,` ... `,` T_n". This usually appears when defining a new expandable mode, where we have:

  "`mode` ⟨*Expandable Mode*⟩ `is` ⟨*Adjective*$_1$⟩ ...⟨*Adjective*$_n$⟩ ⟨*Radix Type*⟩"

This appears to be used only in definitions.

⟨ Classes for type (abstract syntax tree) 798 ⟩ ≡
    { Types }
  RadixTypePtr = ↑*RadixTypeObj*;
  RadixTypeObj = **object** (*TypeExpressionObj*)
    nArgs: *PList*; { of }
    constructor *Init*(**const** *aPos*: *Position*; *aKind*: *TypeSort*; *aArgs*: *PList*);
    destructor *Done*; *virtual*;
  end ;

See also sections 800, 802, 804, and 806.

This code is used in section 737.

799. Constructor.

⟨ Implementing type AST 799 ⟩ ≡
constructor *RadixTypeObj.Init*(**const** *aPos*: *Position*; *aKind*: *TypeSort*; *aArgs*: *PList*);
  begin *nTypePos* ← *aPos*; *nTypeSort* ← *aKind*; *nArgs* ← *aArgs*;
  end;
destructor *RadixTypeObj.Done*;
  begin *dispose*(*nArgs*, *Done*);
  end;

See also sections 801, 803, 805, and 807.

This code is used in section 736.

800. 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".

⟨ Classes for type (abstract syntax tree) 798 ⟩ +≡
  StandardTypePtr = ↑*StandardTypeObj*;
  StandardTypeObj = **object** (*RadixTypeObj*)
    nModeSymbol: *integer*;
    constructor *Init*(**const** *aPos*: *Position*; *aModeSymbol*: *integer*; *aArgs*: *PList*);
    destructor *Done*; *virtual*;
  end ;

801. Constructor.

⟨ Implementing type AST 799 ⟩ +≡
constructor *StandardTypeObj.Init*(**const** *aPos*: *Position*; *aModeSymbol*: *integer*; *aArgs*: *PList*);
  begin *inherited Init*(*aPos*, *wsStandardType*, *aArgs*); *nModeSymbol* ← *aModeSymbol*;
  end;
destructor *StandardTypeObj.Done*;
  begin *inherited Done*;
  end;

802. Structure type. When we define a new structure, we are really introducing a new type. ⟦The *aArgs* tracks its parent structures and parameter types.⟧ The structure type extends the RadixType class because RadixType instances can be "stacked with adjectives".

⟨ Classes for type (abstract syntax tree) 798 ⟩ +≡
  StructTypePtr = ↑*StructTypeObj*;
  StructTypeObj = **object** (*RadixTypeObj*)
    nStructSymbol: *integer*;
    constructor *Init*(**const** *aPos*: *Position*; *aStructSymbol*: *integer*; *aArgs*: *PList*);
    destructor *Done*; *virtual*;
  end ;

803. Constructor.

⟨ Implementing type AST 799 ⟩ +≡
constructor *StructTypeObj.Init*(**const** *aPos*: *Position*; *aStructSymbol*: *integer*; *aArgs*: *PList*);
  begin *inherited Init*(*aPos*, *wsStructureType*, *aArgs*); *nStructSymbol* ← *aStructSymbol*;
  end;
destructor *StructTypeObj.Done*;
  begin *inherited Done*;
  end;

804. Clustered type. The clustered type describes the situation where we accumulate *aCluster* of adjectives atop *aType*.

⟨ Classes for type (abstract syntax tree) 798 ⟩ +≡
  ClusteredTypePtr = ↑*ClusteredTypeObj*;
  ClusteredTypeObj = **object** (*TypeExpressionObj*)
    nAdjectiveCluster: *PList*;
    nType: *TypePtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aCluster*: *PList*; *aType*: *TypePtr*);
    destructor *Done*; *virtual*;
  end ;

805. Constructor.

⟨ Implementing type AST 799 ⟩ +≡
constructor *ClusteredTypeObj.Init*(**const** *aPos*: *Position*; *aCluster*: *PList*; *aType*: *TypePtr*);
  begin *nTypePos* ← *aPos*; *nTypeSort* ← *wsClusteredType*; *nAdjectiveCluster* ← *aCluster*;
  nType ← *aType*;
  end;
destructor *ClusteredTypeObj.Done*;
  begin *dispose*(*nAdjectiveCluster*, *Done*); *dispose*(*nType*, *Done*);
  end;

806. Incorrect type. We want Mizar to be resilient against typing errors, so we have an *IncorrectType* node for the syntax tree. The alternative would be to crash upon error.

⟨ Classes for type (abstract syntax tree) 798 ⟩ +≡
  IncorrectTypePtr = ↑*IncorrectTypeObj*;
  IncorrectTypeObj = **object** (*TypeExpressionObj*)
    constructor *Init*(**const** *aPos*: *Position*);
  end

807. Constructor.

⟨ Implementing type AST 799 ⟩ +≡
constructor *IncorrectTypeObj.Init*(**const** *aPos*: *Position*);
  begin *nTypePos* ← *aPos*; *nTypeSort* ← *wsErrorType*;
  end;

Section 20.3. FORMULAS (ABSTRACT SYNTAX TREE)

808. We have an abstract base class for formulas.

⟨ Abstract base class for formulas 808 ⟩ ≡

FormulaSort = (*wsErrorFormula*, *wsThesis*, *wsContradiction*, *wsRightSideOfPredicativeFormula*, *wsPredicativeFormula*, *wsMultiPredicativeFormula*, *wsPrivatePredicateFormula*, *wsAttributiveFormula*, *wsQualifyingFormula*, *wsUniversalFormula*, *wsExistentialFormula*, *wsNegatedFormula*, *wsConjunctiveFormula*, *wsDisjunctiveFormula*, *wsConditionalFormula*, *wsBiconditionalFormula*, *wsFlexaryConjunctiveFormula*, *wsFlexaryDisjunctiveFormula*);
FormulaPtr = ↑*FormulaExpressionObj*;
FormulaExpressionObj = **object** (*MObject*)
  nFormulaSort: *FormulaSort*;
  nFormulaPos: *Position*;
  end

This code is used in section 737.

809. The syntax for Mizar formulas looks like:

```
Formula-Expression = "(" Formula-Expression ")"
    | Atomic-Formula-Expression
    | Quantified-Formula-Expression
    | Formula-Expression "&" Formula-Expression
    | Formula-Expression "&" "..." "&" Formula-Expression
    | Formula-Expression "or" Formula-Expression
    | Formula-Expression "or" "..." "or" Formula-Expression
    | Formula-Expression "implies" Formula-Expression
    | Formula-Expression "iff" Formula-Expression
    | "not" Formula-Expression
    | "contradiction"
    | "thesis" .
Atomic-Formula-Expression =
    [Term-Expression-List] [("does" | "do") "not"] Predicate-Symbol [Term-Expression-List]
    {[("does" | "do") "not"] Predicate-Symbol Term-Expression-List}
    | Predicate-Identifier "[" [ Term-Expression-List ] "]"
    | Term-Expression "is" Adjective { Adjective }
    | Term-Expression "is" Type-Expression .
Quantified-Formula-Expression =
        "for" Qualified-Variables
        [ "st" Formula-Expression ]
        ( "holds" Formula-Expression | Quantified-Formula-Expression )
    | "ex" Qualified-Variables "st" Formula-Expression .
```

810. Right-side of predicative formula.

⟨ Classes for formula (abstract syntax tree) 810 ⟩ ≡
  { Formulas }
RightSideOfPredicativeFormulaPtr = ↑*RightSideOfPredicativeFormulaObj*;
RightSideOfPredicativeFormulaObj = **object** (*FormulaExpressionObj*)
  nPredNr: *integer*;
  nRightArgs: *PList*;
  constructor *Init*(**const** *aPos*: *Position*; *aPredNr*: *integer*; *aRightArgs*: *PList*);
  destructor *Done*; *virtual*;
end

See also sections 812, 814, 816, 818, 820, 822, 824, 826, 828, 830, 832, 834, 836, 838, 840, 842, 844, 846, and 848.

This code is used in section 737.

811. Constructor.

⟨ Implementing formula AST 811 ⟩ ≡
constructor *RightSideOfPredicativeFormulaObj.Init*(**const** *aPos*: *Position*;
    aPredNr: *integer*;
    aRightArgs: *PList*);
  begin *nFormulaPos* ← *aPos*; *nFormulaSort* ← *wsRightSideOfPredicativeFormula*;
  nPredNr ← *aPredNr*; *nRightArgs* ← *aRightArgs*;
  end;
destructor *RightSideOfPredicativeFormulaObj.Done*;
  begin *dispose*(*nRightArgs*, *Done*);
  end;

See also sections 813, 815, 817, 819, 821, 823, 825, 827, 829, 831, 833, 835, 837, 839, 841, 843, 845, 847, and 849.

This code is used in section 736.

812. Predicative formula. A "predicative" formula refers to a formula involving predicates. A predicate will have a list of terms $\vec{t}$ it expects as arguments, as well as two numbers ℓ, r such that $t_1, \ldots, t_\ell$ are the arguments to its left, and $t_{\ell+1}$, …, $t_{\ell+r}$ are on the right. When $\ell = 0$, all arguments are on the right; and when $r = 0$, all arguments are on the left.

⟨ Classes for formula (abstract syntax tree) 810 ⟩ +≡
PredicativeFormulaPtr = ↑*PredicativeFormulaObj*;
PredicativeFormulaObj = **object** (*RightSideOfPredicativeFormulaObj*)
  nLeftArgs: *PList*;
  constructor *Init*(**const** *aPos*: *Position*; *aPredNr*: *integer*; *aLeftArgs*, *aRightArgs*: *PList*);
  destructor *Done*; *virtual*;
end

813. Constructor.

⟨ Implementing formula AST 811 ⟩ +≡
constructor *PredicativeFormulaObj.Init*(**const** *aPos*: *Position*;
    aPredNr: *integer*;
    aLeftArgs, *aRightArgs*: *PList*);
  begin *nFormulaPos* ← *aPos*; *nFormulaSort* ← *wsPredicativeFormula*; *nPredNr* ← *aPredNr*;
  nLeftArgs ← *aLeftArgs*; *nRightArgs* ← *aRightArgs*;
  end;
destructor *PredicativeFormulaObj.Done*;
  begin *dispose*(*nLeftArgs*, *Done*); *dispose*(*nRightArgs*, *Done*);
  end;

814. Multi-predicative formula. The Working Mathematician writes things like "$1 \leq i \leq \|T\|$" and Mizar wants to support this. Multi-predicative formulas are of this form "`1 <= i <= len T`". This occurs in `VECTSP13`, for example.

⟨ Classes for formula (abstract syntax tree) 810 ⟩ +≡
  MultiPredicativeFormulaPtr = ↑*MultiPredicativeFormulaObj*;
  MultiPredicativeFormulaObj = **object** (*FormulaExpressionObj*)
    nScraps: *PList*;
    constructor *Init*(**const** *aPos*: *Position*; *aScraps*: *PList*);
    destructor *Done*; *virtual*;
  end

815. Constructor.

⟨ Implementing formula AST 811 ⟩ +≡
constructor *MultiPredicativeFormulaObj.Init*(**const** *aPos*: *Position*; *aScraps*: *PList*);
  begin *nFormulaPos* ← *aPos*; *nFormulaSort* ← *wsMultiPredicativeFormula*; *nScraps* ← *aScraps*;
  end;
destructor *MultiPredicativeFormulaObj.Done*;
  begin *dispose*(*nScraps*, *Done*);
  end;

816. Attributive formula. As part of Mizar's soft type system, we can use attributes (adjectives) to form a formula like "⟨*term*⟩ `is` ⟨*adjective*⟩". We can stack multiple adjectives in an attributive formula.

⟨ Classes for formula (abstract syntax tree) 810 ⟩ +≡
  AttributiveFormulaPtr = ↑*AttributiveFormulaObj*;
  AttributiveFormulaObj = **object** (*FormulaExpressionObj*)
    nSubject: *TermPtr*;
    nAdjectives: *PList*;
    constructor *Init*(**const** *aPos*: *Position*; *aSubject*: *TermPtr*; *aAdjectives*: *PList*);
    destructor *Done*; *virtual*;
  end

817. Constructor.

⟨ Implementing formula AST 811 ⟩ +≡
constructor *AttributiveFormulaObj.Init*(**const** *aPos*: *Position*; *aSubject*: *TermPtr*; *aAdjectives*: *PList*);
  begin *nFormulaPos* ← *aPos*; *nFormulaSort* ← *wsAttributiveFormula*; *nSubject* ← *aSubject*;
  nAdjectives ← *aAdjectives*;
  end;
destructor *AttributiveFormulaObj.Done*;
  begin *dispose*(*nSubject*, *Done*); *dispose*(*nAdjectives*, *Done*);
  end;

818. Private predicative formula. When we have "`defpred P[...] means ...`" in Mizar, we refer to "`P`" as a private predicate. It is represented in the abstract syntax tree as a private predicative formula object.

⟨ Classes for formula (abstract syntax tree) 810 ⟩ +≡
  PrivatePredicativeFormulaPtr = ↑*PrivatePredicativeFormulaObj*;
  PrivatePredicativeFormulaObj = **object** (*FormulaExpressionObj*)
    nPredIdNr: *integer*;
    nArgs: *PList*;
    constructor *Init*(**const** *aPos*: *Position*; *aPredIdNr*: *integer*; *aArgs*: *PList*);
    destructor *Done*; *virtual*;
  end

819. Constructor.

⟨ Implementing formula AST 811 ⟩ +≡
constructor *PrivatePredicativeFormulaObj.Init*(**const** *aPos*: *Position*;
    aPredIdNr: *integer*;
    aArgs: *PList*);
  begin *nFormulaPos* ← *aPos*; *nFormulaSort* ← *wsPrivatePredicateFormula*; *nPredIdNr* ← *aPredIdNr*;
  nArgs ← *aArgs*;
  end;
destructor *PrivatePredicativeFormulaObj.Done*;
  begin *dispose*(*nArgs*, *Done*);
  end;

820. Qualifying formula. Using Mizar's soft type system, we may have formulas of the form "⟨*term*⟩ `is` ⟨*type*⟩". These are referred to as "qualifying formulas", at least when discussing the abstract syntax tree.

⟨ Classes for formula (abstract syntax tree) 810 ⟩ +≡
  QualifyingFormulaPtr = ↑*QualifyingFormulaObj*;
  QualifyingFormulaObj = **object** (*FormulaExpressionObj*)
    nSubject: *TermPtr*;
    nType: *TypePtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aSubject*: *TermPtr*; *aType*: *TypePtr*); *y*
    destructor *Done*; *virtual*;
  end

821. Constructor.

⟨ Implementing formula AST 811 ⟩ +≡
constructor *QualifyingFormulaObj.Init*(**const** *aPos*: *Position*; *aSubject*: *TermPtr*; *aType*: *TypePtr*);
  begin *nFormulaPos* ← *aPos*; *nFormulaSort* ← *wsQualifyingFormula*; *nSubject* ← *aSubject*;
  nType ← *aType*;
  end;
destructor *QualifyingFormulaObj.Done*;
  begin *dispose*(*nSubject*, *Done*); *dispose*(*nType*, *Done*);
  end;

822. 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.

⟨ Classes for formula (abstract syntax tree) 810 ⟩ +≡
 NegativeFormulaPtr = ↑*NegativeFormulaObj*;
 NegativeFormulaObj = **object** (*FormulaExpressionObj*)
  nArg: *FormulaPtr*;
  constructor *Init*(**const** *aPos*: *Position*; *aArg*: *FormulaPtr*);
  destructor *Done*; *virtual*;
 end

823. Constructor.

⟨ Implementing formula AST 811 ⟩ +≡
constructor *NegativeFormulaObj.Init*(**const** *aPos*: *Position*; *aArg*: *FormulaPtr*);
 begin *nFormulaPos* ← *aPos*; *nFormulaSort* ← *wsNegatedFormula*; *nArg* ← *aArg*;
 end;
destructor *NegativeFormulaObj.Done*;
 begin *dispose*(*nArg*, *Done*);
 end;

824. Binary arguments formula. We have a class describing formulas involving binary logical connectives. We will extend it to describe conjunctive formulas, disjunctive formulas, conditionals, biconditionals, etc.

⟨ Classes for formula (abstract syntax tree) 810 ⟩ +≡
 BinaryFormulaPtr = ↑*BinaryArgumentsFormula*;
 BinaryArgumentsFormula = **object** (*FormulaExpressionObj*)
  nLeftArg, *nRightArg*: *FormulaPtr*;
  constructor *Init*(**const** *aPos*: *Position*; *aLeftArg*, *aRightArg*: *FormulaPtr*);
  destructor *Done*; *virtual*;
 end

825. Constructor.

⟨ Implementing formula AST 811 ⟩ +≡
constructor *BinaryArgumentsFormula.Init*(**const** *aPos*: *Position*; *aLeftArg*, *aRightArg*: *FormulaPtr*);
 begin *nFormulaPos* ← *aPos*; *nLeftArg* ← *aLeftArg*; *nRightArg* ← *aRightArg*;
 end;
destructor *BinaryArgumentsFormula.Done*;
 begin *dispose*(*nLeftArg*, *Done*); *dispose*(*nRightArg*, *Done*);
 end;

826. Conjunctive formula. A conjunctive formula looks like $\varphi\wedge\psi$ where φ and ψ are logical formulas.

⟨ Classes for formula (abstract syntax tree) 810 ⟩ +≡
 ConjunctiveFormulaPtr = ↑*ConjunctiveFormulaObj*;
 ConjunctiveFormulaObj = **object** (*BinaryArgumentsFormula*)
  constructor *Init*(**const** *aPos*: *Position*; *aLeftArg*, *aRightArg*: *FormulaPtr*);
 end

827. Constructor.

⟨ Implementing formula AST 811 ⟩ +≡
constructor *ConjunctiveFormulaObj.Init*(**const** *aPos*: *Position*; *aLeftArg*, *aRightArg*: *FormulaPtr*);
  begin *inherited Init*(*aPos*, *aLeftArg*, *aRightArg*); *nFormulaSort* ← *wsConjunctiveFormula*;
  end;

828. Disjunctive formula. Disjunctive formulas look like $\varphi \lor \psi$ where φ and ψ are formulas.

⟨ Classes for formula (abstract syntax tree) 810 ⟩ +≡
  DisjunctiveFormulaPtr = ↑*DisjunctiveFormulaObj*;
  DisjunctiveFormulaObj = **object** (*BinaryArgumentsFormula*)
    constructor *Init*(**const** *aPos*: *Position*; *aLeftArg*, *aRightArg*: *FormulaPtr*);
  end

829. Constructor.

⟨ Implementing formula AST 811 ⟩ +≡
constructor *DisjunctiveFormulaObj.Init*(**const** *aPos*: *Position*;
  aLeftArg, *aRightArg*: *FormulaPtr*);
  begin *inherited Init*(*aPos*, *aLeftArg*, *aRightArg*); *nFormulaSort* ← *wsDisjunctiveFormula*;
  end;

830. Conditional formula. Conditional formulas look like $\varphi \implies \psi$ where φ and ψ are formulas.

⟨ Classes for formula (abstract syntax tree) 810 ⟩ +≡
  ConditionalFormulaPtr = ↑*ConditionalFormulaObj*;
  ConditionalFormulaObj = **object** (*BinaryArgumentsFormula*)
    constructor *Init*(**const** *aPos*: *Position*; *aLeftArg*, *aRightArg*: *FormulaPtr*);
  end

831. Constructor.

⟨ Implementing formula AST 811 ⟩ +≡
constructor *ConditionalFormulaObj.Init*(**const** *aPos*: *Position*; *aLeftArg*, *aRightArg*: *FormulaPtr*);
  begin *inherited Init*(*aPos*, *aLeftArg*, *aRightArg*); *nFormulaSort* ← *wsConditionalFormula*;
  end;

832. Biconditional formula. Biconditional formulas look like $\varphi \iff \psi$ where φ and ψ are formulas.

⟨ Classes for formula (abstract syntax tree) 810 ⟩ +≡
  BiconditionalFormulaPtr = ↑*BiconditionalFormulaObj*;
  BiconditionalFormulaObj = **object** (*BinaryArgumentsFormula*)
    constructor *Init*(**const** *aPos*: *Position*; *aLeftArg*, *aRightArg*: *FormulaPtr*);
  end

833. Constructor.

⟨ Implementing formula AST 811 ⟩ +≡
constructor *BiconditionalFormulaObj.Init*(**const** *aPos*: *Position*; *aLeftArg*, *aRightArg*: *FormulaPtr*);
  begin *inherited Init*(*aPos*, *aLeftArg*, *aRightArg*); *nFormulaSort* ← *wsBiconditionalFormula*;
  end;

834. Flexary Conjunctive formula. Flexary conjunctive formulas are unique to Mizar, though the Working Mathematician would recognize them as "just a bunch of conjunctions". These look like $\varphi[1] \wedge \cdots \wedge \varphi[n]$ where $\varphi[i]$ is a formula parametrized by a natural number i.

⟨ Classes for formula (abstract syntax tree) 810 ⟩ +≡
  FlexaryConjunctiveFormulaPtr = ↑*FlexaryConjunctiveFormulaObj*;
  FlexaryConjunctiveFormulaObj = **object** (*BinaryArgumentsFormula*)
    constructor *Init*(**const** *aPos*: *Position*; *aLeftArg*, *aRightArg*: *FormulaPtr*);
  end

835. Constructor.

⟨ Implementing formula AST 811 ⟩ +≡
constructor *FlexaryConjunctiveFormulaObj.Init*(**const** *aPos*: *Position*;
    aLeftArg, *aRightArg*: *FormulaPtr*);
  begin *inherited Init*(*aPos*, *aLeftArg*, *aRightArg*); *nFormulaSort* ← *wsFlexaryConjunctiveFormula*;
  end;

836. Flexary Disjunctive formula. Flexary disjunctive formulas are unique to Mizar, though the Working Mathematician would recognize them as "just a bunch of disjunctions". These look like $\varphi[1] \vee \cdots \vee \varphi[n]$ where $\varphi[i]$ is a formula parametrized by a natural number i.

⟨ Classes for formula (abstract syntax tree) 810 ⟩ +≡
  FlexaryDisjunctiveFormulaPtr = ↑*FlexaryDisjunctiveFormulaObj*;
  FlexaryDisjunctiveFormulaObj = **object** (*BinaryArgumentsFormula*)
    constructor *Init*(**const** *aPos*: *Position*; *aLeftArg*, *aRightArg*: *FormulaPtr*);
  end

837. Constructor.

⟨ Implementing formula AST 811 ⟩ +≡
constructor *FlexaryDisjunctiveFormulaObj.Init*(**const** *aPos*: *Position*;
    aLeftArg, *aRightArg*: *FormulaPtr*);
  begin *inherited Init*(*aPos*, *aLeftArg*, *aRightArg*); *nFormulaSort* ← *wsFlexaryDisjunctiveFormula*;
  end;

838. Quantified formula. First-order logic is distinguished by the use of terms and quantifying formulas over terms. We have a base class for quantified formulas. Using the Mizar soft type system, quantified variables are "qualified segments".

⟨ Classes for formula (abstract syntax tree) 810 ⟩ +≡
  QuantifiedFormulaPtr = ↑*QuantifiedFormulaObj*;
  QuantifiedFormulaObj = **object** (*FormulaExpressionObj*)
    nSegment: *QualifiedSegmentPtr*;
    nScope: *FormulaPtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aSegment*: *QualifiedSegmentPtr*; *aScope*: *FormulaPtr*);
    destructor *Done*; *virtual*;
  end

839. Constructor.

⟨ Implementing formula AST 811 ⟩ +≡

```
constructor QuantifiedFormulaObj.Init(const aPos: Position;
                                      aSegment: QualifiedSegmentPtr;
                                      aScope: FormulaPtr);
  begin nFormulaPos ← aPos; nSegment ← aSegment; nScope ← aScope;
  end;
destructor QuantifiedFormulaObj.Done;
  begin dispose(nSegment, Done); dispose(nScope, Done);
  end;
```

840. Universal formula. When we want to describe a formula of the form "$\forall x : T.\, \varphi[x]$" where T is a soft type and $\varphi[x]$ is a formula parametrized by x.

⟨ Classes for formula (abstract syntax tree) 810 ⟩ +≡

```
  UniversalFormulaPtr = ↑UniversalFormulaObj;
  UniversalFormulaObj = object (QuantifiedFormulaObj)
    constructor Init(const aPos: Position; aSegment: QualifiedSegmentPtr; aScope: FormulaPtr);
  end
```

841. Constructor.

⟨ Implementing formula AST 811 ⟩ +≡

```
constructor UniversalFormulaObj.Init(const aPos: Position;
                                     aSegment: QualifiedSegmentPtr;
                                     aScope: FormulaPtr);
  begin inherited Init(aPos, aSegment, aScope); nFormulaSort ← wsUniversalFormula;
  end;
```

842. Existential formula. The other quantified formula are existentially quantified formulas, which resemble "$\exists x : T.\, \varphi[x]$" where T is a soft type and $\varphi[x]$ is a formula parametrized by x.

⟨ Classes for formula (abstract syntax tree) 810 ⟩ +≡

```
  ExistentialFormulaPtr = ↑ExistentialFormulaObj;
  ExistentialFormulaObj = object (QuantifiedFormulaObj)
    constructor Init(const aPos: Position; aSegment: QualifiedSegmentPtr; aScope: FormulaPtr);
  end
```

843. Constructor.

⟨ Implementing formula AST 811 ⟩ +≡

```
constructor ExistentialFormulaObj.Init(const aPos: Position;
                                       aSegment: QualifiedSegmentPtr;
                                       aScope: FormulaPtr);
  begin inherited Init(aPos, aSegment, aScope); nFormulaSort ← wsExistentialFormula;
  end;
```

844. Contradiction formula. The canonical contradiction $\perp$ in Mizar is represented by the reserved keyword "`contradiction`".

⟨ Classes for formula (abstract syntax tree) 810 ⟩ +≡

```
  ContradictionFormulaPtr = ↑ContradictionFormulaObj;
  ContradictionFormulaObj = object (FormulaExpressionObj)
    constructor Init(const aPos: Position);
  end
```

845. Constructor.

⟨ Implementing formula AST 811 ⟩ +≡
constructor *ContradictionFormulaObj.Init*(**const** *aPos*: *Position*);
  begin *nFormulaPos* ← *aPos*; *nFormulaSort* ← *wsContradiction*;
  end;

846. Thesis formula. When we are in the middle of a proof, the goal or obligation left to be proven is called the "`thesis`".

⟨ Classes for formula (abstract syntax tree) 810 ⟩ +≡
  ThesisFormulaPtr = ↑*ThesisFormulaObj*;
  ThesisFormulaObj = **object** (*FormulaExpressionObj*)
    constructor *Init*(**const** *aPos*: *Position*);
  end

847. Constructor.

⟨ Implementing formula AST 811 ⟩ +≡
constructor *ThesisFormulaObj.Init*(**const** *aPos*: *Position*);
  begin *nFormulaPos* ← *aPos*; *nFormulaSort* ← *wsThesis*;
  end;

848. Incorrect formula. We also have a node in abstract syntax trees for "incorrect" formulas.

⟨ Classes for formula (abstract syntax tree) 810 ⟩ +≡
  IncorrectFormulaPtr = ↑*IncorrectFormula*;
  IncorrectFormula = **object** (*FormulaExpressionObj*)
    constructor *Init*(**const** *aPos*: *Position*);
  end

849. Constructor.

⟨ Implementing formula AST 811 ⟩ +≡
constructor *IncorrectFormula.Init*(**const** *aPos*: *Position*);
  begin *nFormulaPos* ← *aPos*; *nformulaSort* ← *wsErrorFormula*;
  end;

File 21

Weakly strict Mizar article

850. The parser "eats in" a mizar article, then produces a `.wsx` (weakly strict Mizar) XML file containing the abstract syntax tree, and also a `.frt` article containing the formats for the article.

This strategy should be familiar to anyone who has looked into compilers and interpreters: transform the abstract syntax tree into an intermediate representation, then transform the intermediate representations in various passes.

This module will transform the parse tree to an abstract syntax tree in XML format.

⟨ wsmarticle.pas 850 ⟩ ≡
  ⟨ GNU License 4 ⟩
unit *wsmarticle*;
  interface
  uses *mobjects*, *errhan*, *mscanner*, *syntax*, *abstract_syntax*, *xml_dict*, *xml_inout*;
    ⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩
  const
    ⟨ Publicly declared constants in `wsmarticle.pas` 855 ⟩
    ⟨ Publicly declared functions in `wsmarticle.pas` 853 ⟩
    ⟨ Global variables publicly declared in `wsmarticle.pas` 1007 ⟩
  implementation
  uses *mizenv*, *mconsole*, *librenv*, *scanner*, *xml_parser*
      mdebug , *info* **end_mdebug**;
    ⟨ Implementation for `wsmarticle.pas` 854 ⟩
  end .

851. Exercise. We will create a class hierarchy for the abstract syntax trees for Mizar. A lot of this is boiler-plate. The reader is invited to write a couple of programs which will:
(1) read in an EBNF-like grammar and emit the class hierarchy for its abstract syntax tree.
(2) read in an EBNF-like grammar, and emit the class hierarchy for generating the XML for it.

After all, if you look at the sheer number of sections in this file, it's staggeringly huge. But a lot of it is boiler-plate.

852. ⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ ≡
See also sections 856, 862, 864, 867, 868, 870, 871, 875, 877, 879, 881, 884, 886, 888, 890, 893, 895, 897, 900, 906, 909, 911, 913, 915, 916, 924, 926, 928, 930, 946, 948, 950, 952, 954, 956, 958, 960, 962, 964, 966, 968, 970, 972, 974, 976, 978, 980, 982, 984, 987, 989, 991, 993, 995, 997, 999, 1001, 1003, 1014, 1097, and 1148.
This code is used in section 850.

853. ⟨ Publicly declared functions in `wsmarticle.pas` 853 ⟩ ≡
This code is used in section 850.

854. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ ≡

See also sections 857, 858, 861, 863, 865, 869, 872, 876, 878, 880, 882, 885, 887, 889, 891, 894, 896, 898, 901, 907, 910, 912, 914, 917, 918, 925, 927, 929, 931, 934, 936, 938, 940, 942, 947, 949, 951, 953, 955, 957, 959, 961, 963, 965, 967, 969, 971, 973, 975, 977, 979, 981, 983, 985, 988, 990, 992, 994, 996, 998, 1000, 1002, 1004, 1005, 1006, 1008, 1013, 1015, 1016, 1017, 1018, 1019, 1020, 1021, 1022, 1023, 1024, 1025, 1026, 1027, 1042, 1043, 1044, 1045, 1058, 1059, 1060, 1061, 1067, 1068, 1069, 1070, 1071, 1072, 1073, 1074, 1075, 1076, 1077, 1078, 1079, 1080, 1095, 1098, 1099, 1100, 1101, 1102, 1103, 1104, 1105, 1106, 1107, 1108, 1109, 1110, 1111, 1112, 1113, 1114, 1115, 1118, 1119, 1120, 1121, 1122, 1123, 1124, 1125, 1126, 1127, 1128, 1129, 1130, 1131, 1132, 1133, 1134, 1135, 1136, 1137, 1138, 1139, 1140, 1141, 1142, 1143, 1144, 1145, 1146, 1147, 1149, 1150, 1151, 1152, 1153, 1154, 1155, 1156, 1157, 1158, 1159, 1160, 1161, 1162, 1163, 1164, 1165, 1166, 1167, 1168, 1169, 1170, 1171, 1172, 1173, 1174, 1175, 1176, 1177, 1178, 1179, 1180, 1181, 1182, 1183, 1184, 1185, 1186, 1187, 1188, 1189, 1190, 1191, and 1192.

This code is used in section 850.

Section 21.1. WEAKLY STRICT TEXT PROPER

855. Mizar provides a grammar for its syntax in the file `/usr/local/doc/mizar/syntax`
It uses a variant of EBNF:
- Terminal symbols are written `"in quotes"`
- Production rules are separated by vertical lines "|"
- Optional symbols are placed in `[brackets]`
- Repeated items zero or more times are placed in `{braces}`.
- Rules end in a period "."

We will freely quote from `syntax.txt`, rearranging the rules as needed to discuss the relevant parts of Mizar's grammar. We will write the `syntax.txt` passages in typewriter font.

We should recall the syntax for text items:

```
Text-Proper = Section { Section } .
Section = "begin" { Text-Item } .
Text-Item = Reservation
  | Definitional-Item
  | Registration-Item
  | Notation-Item
  | Theorem
  | Scheme-Item
  | Auxiliary-Item .
Definitional-Item = Definitional-Block ";" .
Registration-Item = Registration-Block ";" .
Theorem = "theorem" Compact-Statement .
Compact-Statement = Proposition Justification ";" .
Justification = Simple-Justification | Proof .
Auxiliary-Item = Statement | Private-Definition .
```

These are the different syntactic classes for "top-level statements" in the text (not the environment header) of a Mizar article. The interested reader can investigate the `syntax.txt` file more fully to get all the block statements in Mizar. We have already made these different kinds of blocks syntactic values of *BlockKind* earlier (§692). Now we want to be able to translate them into English. We will just skip ahead and make these different syntactic classes into values of an enumerated type.

⟨ Publicly declared constants in `wsmarticle.pas` 855 ⟩ ≡
BlockName: **array** [*BlockKind*] **of** *string* =
  (´`Text-Proper`´, { blMain }
  ´`Now-Reasoning`´, { blDiffuse }
  ´`Hereby-Reasoning`´, { blHereby }
  ´`Proof`´, { blProof }
  ´`Definitional-Block`´, { blDefinition }
  ´`Notation-Block`´, { blNotation }
  ´`Registration-Block`´, { blRegistration }
  ´`Case`´, { blCase }
  ´`Suppose`´, { blSuppose }
  ´`Scheme-Block`´ { blPublicScheme });

This code is used in section 850.

856. Class hierarchy for blocks. We can now translate the grammar for blocks into a class hierarchy. The "text proper" extends an abstract "block" statement. We will provide factory methods "*wsTextProper.NewBlock*" and "*NewItem*" for adding a new block (and item) contained within the caller "block". We will be tracking the "kind" of block (§692), and the text proper will need to track which article it belongs to.

All the various kinds of blocks are handled with this one class: proofs, definitions, notations, registrations, cases, suppose blocks, schemes, hereby statements, and so on. However, some of these blocks have extra content which needs their own nodes in the abstract syntax tree, especially Definitions (§§943 *et seq.*) and Registrations (§§986 *et seq.*).

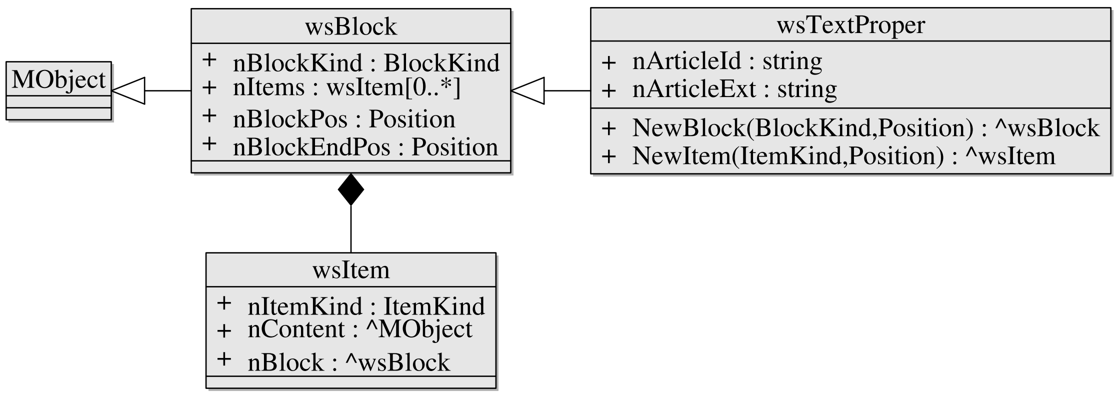

Fig. 4. UML class diagram for *wsBlock* and related classes.

It is important to stress: **wsBlock instances represent all statements which are block statements and all other statements are wsItem instances.** Looking back at the different kinds of blocks, you see that they are "block openers" and will expect to have a matching "`end`" statement closing it.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡

```
wsBlockPtr = ↑wsBlock;
wsBlock = object (MObject)
  nBlockKind: BlockKind;
  nItems: PList;   { list of wsItem objects }
  nBlockPos, nBlockEndPos: Position;
  constructor Init(aBlokKind : BlockKind ; const aPos: Position);
  destructor Done; virtual;
end ;
⟨ Weakly strict Item class 860 ⟩;
wsTextProperPtr = ↑wsTextProper;
wsTextProper = object (wsBlock)
  nArticleID, nArticleExt: string;
  constructor Init(const aArticleID, aArticleExt: string ; const aPos: Position);
  destructor Done; virtual;
  function NewBlock(aBlockKind : BlockKind ; const aPos: Position): wsBlockPtr;
  function NewItem(aItemKind : ItemKind ; const aPos: Position): wsItemPtr;
end ;
```

857. Constructor. We initialize using the inherited *wsBlock* constructor (§859). The "text proper" refers to a block which is as top-level as possible, so we construct it as a block whose kind is `blMain` located at *aPos*.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *wsTextProper.Init*(**const** *aArticleID*, *aArticleExt*: *string*; **const** *aPos*: *Position*);
  begin *inherited Init*(*blMain*, *aPos*); *nArticleID* ← *aArticleID*; *nArticleExt* ← *aArticleExt*;
  end;

destructor *wsTextProper.Done*;
  begin *inherited Done*;
  end;

858. Adding statements into a block. we will add a block to a "text proper", which will then construct a block which tracks the caller as its containing block. This requires giving the kind of the newly minted block (§692).

Similarly, when constructing an item which is contained in the block, we need to pass along the item kind (§702).

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *wsTextProper.NewBlock*(*aBlockKind* : *BlockKind* ; **const** *aPos*: *Position*): *wsBlockPtr*;
  begin *result* ← *new*(*WSBlockPtr*, *Init*(*aBlockKind*, *CurPos*));
  end;

function *wsTextProper.NewItem*(*aItemKind* : *ItemKind* ; **const** *aPos*: *Position*): *wsItemPtr*;
  begin *result* ← *new*(*wsItemPtr*, *Init*(*aItemKind*, *CurPos*));
  end;

859. Block Constructor. Curiously, the *MObject* constructor (§187) is not invoked when constructing a *wsBlock*. We will also need the position (§103) of the block in the article. The collection of items in the block is initialized to be empty.

constructor *wsBlock.Init*(*aBlokKind* : *BlockKind* ; **const** *aPos*: *Position*);
  begin *nBlockKind* ← *aBlokKind*; *nBlockPos* ← *aPos*; *nBlockEndPos* ← *aPos*;
  nItems ← *New*(*PList*, *Init*(0));
  end;

destructor *wsBlock.Done*;
  begin *dispose*(*nItems*, *Done*); *inherited Done*;
  end;

860. Text items. An item requires its "kind" (§702) for its syntactic class.

⟨ Weakly strict Item class 860 ⟩ ≡
  wsItemPtr = ↑*wsItem*;
  wsItem = **object** (*MObject*)
    nItemKind: *ItemKind*;
    nItemPos, *nItemEndPos*: *Position*;
    nContent: *PObject*;
    nBlock: *wsBlockPtr*;
    constructor *Init*(*aItemKind* : *ItemKind* ; **const** *aPos*: *Position*);
    destructor *Done*; *virtual*;
  end ;

This code is used in section 856.

861. Constructor

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *wsItem.Init*(*aItemKind* : *ItemKind* ; **const** *aPos*: *Position*);
  begin *nItemKind* ← *aItemKind*; *nItemPos* ← *aPos*; *nItemEndPos* ← *aPos*; *nContent* ← **nil**;
  nBlock ← **nil**;
  end;
destructor *wsItem.Done*;
  begin if *nBlock* ≠ **nil then** *dispose*(*nBlock*, *Done*);
  inherited Done;
  end;

862. Pragmas. Mizar supports pragmas (analogous to conditional compilation).

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  PragmaPtr = ↑*PragmaObj*;
  PragmaObj = **object** (*MObject*)
    nPragmaStr: *string*;
    constructor *Init*(*aStr* : *string*);
  end ;

863. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *PragmaObj.Init*(*aStr* : *string*);
  begin *nPragmaStr* ← *aStr*;
  end;

864. Labels and propositions. A proposition is just a sentence with a label. We will need to represent both of these in our abstract syntax tree.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  LabelPtr = ↑*LabelObj*;
  LabelObj = **object** (*MObject*)
    nLabelIdNr: *integer*;
    nLabelPos: *Position*;
    constructor *Init*(*aLabelId* : *integer* ; **const** *aPos*: *Position*);
  end ;
  PropositionPtr = ↑*PropositionObj*;
  PropositionObj = **object** (*mObject*)
    nLab: *LabelPtr*;
    nSntPos: *Position*;
    nSentence: *FormulaPtr*;
    constructor *Init*(*aLab* : *LabelPtr*; *aSentence* : *FormulaPtr* ; **const** *aSntPos*: *Position*);
    destructor *Done*; *virtual*;
  end ;

865. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *LabelObj.Init*(*aLabelId* : *integer* ; **const** *aPos*: *Position*);
  begin *nLabelIdNr* ← *aLabelId*; *nLabelPos* ← *aPos*;
  end;

constructor *PropositionObj.Init*(*alab* : *LabelPtr*; *aSentence* : *FormulaPtr* ; **const** *aSntPos*: *Position*);
  begin *nLab* ← *aLab*; *nSntPos* ← *aSntPos*; *nSentence* ← *aSentence*;
  end;
destructor *PropositionObj.Done*;
  begin *dispose*(*nLab*, *Done*); *dispose*(*nSentence*, *Done*);
  end;

866. References. References are either local (i.e., from the file being processed) or library (i.e., from the Mizar math library). The grammar for library references is rather generous. The basic rules are that we have theorem references,

$$\langle\mathit{article}\rangle{:}\langle\mathit{number}\rangle$$

and definition references,

$$\langle\mathit{article}\rangle{:}\ \texttt{def}\ \ \langle\mathit{number}\rangle$$

and scheme references,

$$\langle\mathit{article}\rangle{:}\ \texttt{sch}\ \ \langle\mathit{number}\rangle$$

What makes it tricky is we also allow multiple references from the same article to just add a comma followed by the theorem number

$$\langle\mathit{article}\rangle{:}\langle\mathit{number}\rangle\{,\langle\mathit{number}\rangle\}$$

or a comma followed by definition numbers

$$\langle\mathit{article}\rangle{:}\ \texttt{def}\ \ \langle\mathit{number}\rangle\{,\texttt{def}\ \ \langle\mathit{number}\rangle\}$$

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:

```
Reference = Local-Reference | Library-Reference .

Scheme-Reference = Local-Scheme-Reference
  | Library-Scheme-Reference .

Local-Reference = Label-Identifier .

Local-Scheme-Reference = Scheme-Identifier .

Library-Reference = Article-Name ":" ( Theorem-Number | "def" Definition-Number )
  { "," ( Theorem-Number | "def" Definition-Number ) } .

Library-Scheme-Reference = Article-Name ":" "sch" Scheme-Number .
```

867. Class structure. We have an abstract "reference" class, which is either a local reference (to a label within the article) or a library reference (to some result in the MML).

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  ReferenceKind = (*LocalReference*, *TheoremReference*, *DefinitionReference*);
  ⟨ Inference kinds (`wsmarticle.pas`) 874 ⟩;
  ReferencePtr = ↑*ReferenceObj*;
  ReferenceObj = **object** (*MObject*)
    nRefSort: *ReferenceKind*;
    nRefPos: *Position*;
  end;

868. Local references.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  LocalReferencePtr = ↑*LocalReferenceObj*;
  LocalReferenceObj = **object** (*ReferenceObj*)
    nLabId: *integer*;
    constructor *Init*(*aLabId* : *integer* ; **const** *aPos*: *Position*);
  end ;

869. Constructor. The reference constructors simply populate the appropriate fields in the reference, and the position in the article's text.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *LocalReferenceObj.Init*(*aLabId* : *integer*;
    const *aPos*: *Position*);
  begin *nRefSort* ← *LocalReference*; *nLabId* ← *aLabId*; *nRefPos* ← *aPos*
  end;

870. Library references. This is the abstract class representing either theorem or definition references from an article.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  LibraryReferencePtr = ↑*LibraryReferenceObj*;
  LibraryReferenceObj = **object** (*ReferenceObj*)
    nArticleNr: *integer*;
  end;

871. 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` 852 ⟩ +≡
  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 ;

872. Constructor. The reference constructors simply populate the appropriate fields in the reference, and the position in the article's text.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *TheoremReferenceObj.Init*(*aArticleNr*, *aTheoNr* : *integer*;
    const *aPos*: *Position*);
  begin *nRefSort* ← *TheoremReference*; *nArticleNr* ← *aArticleNr*; *nTheoNr* ← *aTheoNr*;
  nRefPos ← *aPos*
  end;
constructor *DefinitionReferenceObj.Init*(*aArticleNr*, *aDefNr* : *integer*;
    const *aPos*: *Position*);
  begin *nRefSort* ← *DefinitionReference*; *nArticleNr* ← *aArticleNr*; *nDefNr* ← *aDefNr*;
  nRefPos ← *aPos*
  end;

873. 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.

874. The different kinds of inference, since a *Justification* is a tagged union of sorts.

⟨ Inference kinds (`wsmarticle.pas`) 874 ⟩ ≡
  InferenceKind = (*infError*, *infStraightforwardJustification*, *infSchemeJustification*, *infProof*,
    infSkippedProof)

This code is used in section 867.

875. Class structure for justifications. The class hierarchy for justifications reflects the grammar we just discussed.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  JustificationPtr = ↑*JustificationObj*;
  JustificationObj = **object** (*MObject*)
    nInfSort: *InferenceKind*;
    nInfPos: *Position*;
    constructor *Init*(*aInferSort* : *InferenceKind* ; **const** *aPos*: *Position*);
  end ;

876. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *JustificationObj.Init*(*aInferSort* : *InferenceKind* ; **const** *aPos*: *Position*);
  begin *nInfSort* ← *aInferSort*; *nInfPos* ← *aPos*;
  end;

877. Simple justifications. These are either "`by`" a list of references, or "`from`" a scheme.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  SimpleJustificationPtr = ↑*SimpleJustificationObj*;
  SimpleJustificationObj = **object** (*JustificationObj*)
    nReferences: *PList*;
    constructor *Init*(*aInferSort* : *InferenceKind* ; **const** *aPos*: *Position*);
    destructor *Done*; *virtual*;
  end ;

878. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *SimpleJustificationObj.Init*(*aInferSort* : *InferenceKind* ; **const** *aPos*: *Position*);
  begin *inherited Init*(*aInferSort*, *aPos*); *nReferences* ← *new*(*Plist*, *Init*(0));
  end;

destructor *SimpleJustificationObj.Done*;
  begin *dispose*(*nReferences*, *Done*); *inherited Done*;
  end;

879. Straightforward justification.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  StraightforwardJustificationPtr = ↑*StraightforwardJustificationObj*;
  StraightforwardJustificationObj = **object** (*SimpleJustificationObj*)
    nLinked: *boolean*;
    nLinkPos: *Position*;
    constructor *Init*(**const** *aPos*: *Position*; *aLinked*: *boolean*; **const** *aLinkPos*: *Position*);
    destructor *Done*; *virtual*;
  end ;

880. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *StraightforwardJustificationObj.Init*(**const** *aPos*: *Position*;
aLinked: *boolean*;
const *aLinkPos*: *Position*);
begin *inherited Init*(*infStraightforwardJustification*, *aPos*); *nLinked* ← *aLinked*; *nLinkPos* ← *aLinkPos*;
end;

destructor *StraightforwardJustificationObj.Done*;
begin *inherited Done*;
end;

881. Scheme justification.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
SchemeJustificationPtr = ↑*SchemeJustificationObj*;
SchemeJustificationObj = **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 ;

882. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *SchemeJustificationObj.Init*(**const** *aPos*: *Position*; *aArticleNr*, *aNr*: *integer*);
begin *inherited Init*(*infSchemeJustification*, *aPos*); *nSchFileNr* ← *aArticleNr*; *nSchemeIdNr* ← *aNr*;
nSchemeInfPos ← *aPos*;
end;

destructor *SchemeJustificationObj.Done*;
begin *inherited Done*;
end;

Section 21.2. SCHEMES

883. The grammar for schemes looks like:

```
Scheme-Item = Scheme-Block ";" .
Scheme-Block = "scheme" Scheme-Identifier "{" Scheme-Parameters "}" ":"
  Scheme-Conclusion ["provided" Scheme-Premise {"and" Scheme-Premise}]
  ("proof" | ";") Reasoning "end" .
Scheme-Identifier = Identifier .
Scheme-Parameters = Scheme-Segment  "," Scheme-Segment  .
Scheme-Conclusion = Sentence .
Scheme-Premise = Proposition .
Scheme-Segment = Predicate-Segment | Functor-Segment .
Predicate-Segment =
  Predicate-Identifier {"," Predicate-Identifier} "["[Type-Expression-List] "]" .
Predicate-Identifier = Identifier .
Functor-Segment =
  Functor-Identifier {"," Functor-Identifier} "(" [Type-Expression-List] ")" Specification .
Functor-Identifier = Identifier .
```

We begin with the abstract syntax for scheme parameters.

884. Class hierarchy for schemes. We need "predicate segments" and "functor segments" for the second-order variable parameters to the scheme.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  SchemeSegmentKind = (*PredicateSegment*, *FunctorSegment*);
  SchemeSegmentPtr = ↑*SchemeSegmentObj*;
  SchemeSegmentObj = **object** (*MObject*)
    nSegmPos: *Position*;
    nSegmSort: *SchemeSegmentKind*;
    nVars: *PList*;
    nTypeExpList: *PList*;
    constructor *Init*(**const** *aPos*: *Position*; *aSegmSort*: *SchemeSegmentKind*;
      aVars, *aTypeExpList*: *PList*);
    destructor *Done*; *virtual*;
  end ;

885. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *SchemeSegmentObj.Init*(**const** *aPos*: *Position*;
    aSegmSort: *SchemeSegmentKind*;
    aVars, *aTypeExpList*: *PList*);
  begin *nSegmPos* ← *aPos*; *nSegmSort* ← *aSegmSort*; *nVars* ← *aVars*; *nTypeExpList* ← *aTypeExpList*;
  end;
destructor *SchemeSegmentObj.Done*;
  begin *dispose*(*nVars*, *Done*); *dispose*(*nTypeExpList*, *Done*);
  end;

886. Segment variables for schemes. We need "predicate segments" and "functor segments" for the second-order variable parameters to the scheme.

⟨Publicly declared types in `wsmarticle.pas` 852⟩ +≡
  PredicateSegmentPtr = *SchemeSegmentPtr*;
  FunctorSegmentPtr = ↑*FunctorSegmentObj*;
  FunctorSegmentObj = **object** (*SchemeSegmentObj*)
    nSpecification: *TypePtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aVars*, *aTypeExpList*: *PList*; *aSpecification*: *TypePtr*);
    destructor *Done*; *virtual*;
  end ;

887. Constructor.

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
constructor *FunctorSegmentObj.Init*(**const** *aPos*: *Position*;
    aVars, *aTypeExpList*: *PList*;
    aSpecification: *TypePtr*);
  begin *inherited Init*(*aPos*, *FunctorSegment*, *aVars*, *aTypeExpList*); *nSpecification* ← *aSpecification*;
  end;
destructor *FunctorSegmentObj.Done*;
  begin *dispose*(*nSpecification*, *Done*); *inherited Done*;
  end;

888. 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.

⟨Publicly declared types in `wsmarticle.pas` 852⟩ +≡
  SchemePtr = ↑*SchemeObj*;
  SchemeObj = **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 ;

889. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *SchemeObj.Init*(*aIdNr* : *integer*;
  const *aPos*: *Position*;
  aParams: *PList*;
  aPrems: *PList*;
  aConcl: *FormulaPtr*);
  begin *nSchemeIdNr* ← *aIdNr*; *nSchemePos* ← *aPos*; *nSchemeParams* ← *aParams*;
  nSchemeConclusion ← *aConcl*; *nSchemePremises* ← *aPrems*;
  end;
destructor *SchemeObj.Done*;
  begin *dispose*(*nSchemeParams*, *Done*); *dispose*(*nSchemeConclusion*, *Done*);
  dispose(*nSchemePremises*, *Done*);
  end;

890. Reservations. We can "reserve" an identifier and its type, so we do not need to quantify over it for each theorem. The grammar for it:

```
Reservation = "reserve" Reservation-Segment { "," Reservation-Segment} ";" .
Reservation-Segment = Reserved-Identifiers "for" Type-Expression .
Reserved-Identifiers = Identifier { "," Identifier } .
```

The data needed for a `reserved` node in the abstract syntax tree amounts to a list of identifiers and a type.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  ReservationSegmentPtr = ↑*ReservationSegmentObj*;
  ReservationSegmentObj = **object** (*MObject*)
    nIdentifiers: *PList*;
    nResType: *TypePtr*;
    constructor *Init*(*aIdentifiers* : *PList*; *aType* : *TypePtr*);
    destructor *Done*; *virtual*;
  end ;

891. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *ReservationSegmentObj.Init*(*aIdentifiers* : *PList*; *aType* : *TypePtr*);
  begin *nIdentifiers* ← *aIdentifiers*; *nResType* ← *aType*;
  end;
destructor *ReservationSegmentObj.Done*;
  begin *dispose*(*nIdentifiers*, *Done*); *dispose*(*nResType*, *Done*);
  end;

Section 21.3. PRIVATE DEFINITIONS

892. The grammar for "private definitions" (which introduces block-local or article-local terms and predicates) looks like:

```
Private-Definition = Constant-Definition
  | Private-Functor-Definition
  | Private-Predicate-Definition .
Constant-Definition = "set" Equating-List ";" .
Equating-List = Equating {"," Equating }.
Equating = Variable-Identifier "=" Term-Expression .
Private-Functor-Definition = "deffunc" Private-Functor-Pattern "=" Term-Expression ";" .
Private-Predicate-Definition = "defpred" Private-Predicate-Pattern "means" Sentence ";" .
Private-Functor-Pattern = Functor-Identifier "(" [ Type-Expression-List ] ")" .
Private-Predicate-Pattern = Predicate-Identifier "[" [Type-Expression-List ] "]" .
```

So we really only need to describe private predicates, private functors, and "constant definitions" (which introduce an abbreviation).

893. Private functors.

⟨Publicly declared types in `wsmarticle.pas` 852⟩ +≡
  PrivateFunctorDefinitionPtr = ↑*PrivateFunctorDefinitionObj*;
  PrivateFunctorDefinitionObj = **object** (*MObject*)
    nFuncId: *VariablePtr*;
    nTypeExpList: *PList*;
    nTermExpr: *TermPtr*;
    constructor *Init*(*aFuncId* : *VariablePtr*; *aTypeExpList* : *Plist*; *aTerm* : *TermPtr*);
    destructor *Done*; *virtual*;
  end ;

894. Constructor.

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
constructor *PrivateFunctorDefinitionObj.Init*(*aFuncId* : *VariablePtr*; *aTypeExpList* : *Plist*;
    aTerm : *TermPtr*);
  begin *nFuncId* ← *aFuncId*; *nTypeExpList* ← *aTypeExpList*; *nTermExpr* ← *aTerm*;
  end;
destructor *PrivateFunctorDefinitionObj.Done*;
  begin *dispose*(*nFuncId*, *Done*); *dispose*(*nTypeExpList*, *Done*); *dispose*(*nTermExpr*, *Done*);
  end;

895. Private predicates.

⟨Publicly declared types in `wsmarticle.pas` 852⟩ +≡
  PrivatePredicateDefinitionPtr = ↑*PrivatePredicateDefinitionObj*;
  PrivatePredicateDefinitionObj = **object** (*MObject*)
    nPredId: *VariablePtr*;
    nTypeExpList: *PList*;
    nSentence: *FormulaPtr*;
    constructor *Init*(*aPredId* : *VariablePtr*; *aTypeExpList* : *Plist*; *aSnt* : *FormulaPtr*);
    destructor *Done*; *virtual*;
  end ;

896. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *PrivatePredicateDefinitionObj.Init*(*aPredId* : *VariablePtr*; *aTypeExpList* : *Plist*;
    aSnt : *FormulaPtr*);
  begin *nPredId* ← *aPredId*; *nTypeExpList* ← *aTypeExpList*; *nSentence* ← *aSnt*;
  end;

destructor *PrivatePredicateDefinitionObj.Done*;
  begin *dispose*(*nPredId*, *Done*); *dispose*(*nTypeExpList*, *Done*); *dispose*(*nSentence*, *Done*);
  end;

897. Constant definitions. These are little more than abbreviations for terms, and their implementations reflects this: they are pointers with delusions of grandeur.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  ConstantDefinitionPtr = ↑*ConstantDefinitionObj*;
  ConstantDefinitionObj = **object** (*MObject*)
    nVarId: *VariablePtr*;
    nTermExpr: *TermPtr*;
    constructor *Init*(*aVarId* : *VariablePtr*; *aTerm* : *TermPtr*);
    destructor *Done*; *virtual*;
  end ;

898. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *ConstantDefinitionObj.Init*(*aVarId* : *VariablePtr*; *aTerm* : *TermPtr*);
  begin *nVarId* ← *aVarId*; *nTermExpr* ← *aTerm*;
  end;

destructor *ConstantDefinitionObj.Done*;
  begin *dispose*(*nVarId*, *Done*); *dispose*(*nTermExpr*, *Done*);
  end;

Section 21.4. CHANGING TYPES

899. Each term has a soft type associated with it, but we can "`reconsider`" or change its type. Mizar requires a proof that the term really has the new type. The grammar for this statement:

```
Type-Changing-Statement =
  "reconsider" Type-Change-List "as" Type-ExpressionSimple-Justification ";" .
Type-Change-List =
  (Equating | Variable-Identifier) {"," (Equating | Variable-Identifier)} .
```

This requires a bit of work since we really have *two* types of reconsiderations within a single reconsider statement:

(1) "`reconsider` ⟨*identifier*⟩ `as` ⟨*type*⟩"

(2) "`reconsider` ⟨*identifier*⟩ = ⟨*term*⟩ `as` ⟨*type*⟩"

The trick is to represent a `Type-Change-List` as a list of `Type-Change`s. Then a `Type-Change-Statement` is just a `Type-Change-List` and a type.

900. Class hierarchy.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡

TypeChangeSort = (*Equating*, *VariableIdentifier*);

TypeChangePtr = ↑*TypeChangeObj*;
TypeChangeObj = **object** (*MObject*)
  nTypeChangeKind: *TypeChangeSort*;
  nVar: *VariablePtr*;
  nTermExpr: *TermPtr*;
  constructor *Init*(*aKind* : *TypeChangeSort*; *aVar* : *VariablePtr*; *aTerm* : *TermPtr*);
  destructor *Done*; *virtual*;
end ;

⟨ Example classes (`wsmarticle.pas`) 903 ⟩

TypeChangingStatementPtr = ↑*TypeChangingStatementObj*;
TypeChangingStatementObj = **object** (*MObject*)
  nTypeChangeList: *PList*;
  nTypeExpr: *TypePtr*;
  nJustification: *SimpleJustificationPtr*;
  constructor *Init*(*aTypeChangeList* : *PList*; *aTypeExpr* : *TypePtr*;
      aJustification : *SimpleJustificationPtr*);
  destructor *Done*; *virtual*;
end ;

901. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

constructor *TypeChangeObj.Init*(*aKind* : *TypeChangeSort*; *aVar* : *VariablePtr*; *aTerm* : *TermPtr*);
  begin *nTypeChangeKind* ← *aKind*; *nVar* ← *aVar*; *nTermExpr* ← *aTerm*;
  end;

destructor *TypeChangeObj.Done*;
  begin *dispose*(*nVar*, *Done*);
  if *nTermExpr* ≠ **nil then** *dispose*(*nTermExpr*, *Done*);
  end;

⟨ Constructors for example statements (`wsmarticle.pas`) 904 ⟩

constructor *TypeChangingStatementObj.Init*(*aTypeChangeList* : *PList*; *aTypeExpr* : *TypePtr*;
    aJustification : *SimpleJustificationPtr*);
  begin *nTypeChangeList* ← *aTypeChangeList*; *nTypeExpr* ← *aTypeExpr*;
  nJustification ← *aJustification*;
  end;

destructor *TypeChangingStatementObj.Done*;
  begin *dispose*(*nTypeChangeList*, *Done*); *dispose*(*nTypeExpr*, *Done*); *dispose*(*nJustification*, *Done*);
  end;

Section 21.5. PROOF STEPS

902. Most of the proof steps are handled in generic text-item objects. But there are a few which are outside that tagged union. In particular: existential elimination (`consider` ⟨*variables*⟩ `such that` ⟨*formula*⟩), existential introduction (`take` ⟨*terms*⟩), and concluding statements (`thus` ⟨*formula*⟩).

903. Examples, existential introduction. The proof step "`take` x" transforms goals of the form $\exists x.\, P[x]$ into a new goal $P[x]$. The grammar for examples looks like:

```
Exemplification = "take" Example {"," Example} ";" .
Example = Term-Expression | Variable-Identifier "=" Term-Expression .
```

⟨ Example classes (`wsmarticle.pas`) 903 ⟩ ≡

```
ExamplePtr = ↑ExampleObj;
ExampleObj = object (MObject)
  nVarId: VariablePtr;
  nTermExpr: TermPtr;
  constructor Init(aVarId : VariablePtr; aTerm : TermPtr);
  destructor Done; virtual;
end ;
```

This code is used in section 900.

904. Constructor.

⟨ Constructors for example statements (`wsmarticle.pas`) 904 ⟩ ≡

```
constructor ExampleObj.Init(aVarId : VariablePtr; aTerm : TermPtr);
  begin nVarId ← aVarId; nTermExpr ← aTerm;
  end;
destructor ExampleObj.Done;
  begin if nVarId ≠ nil then dispose(nVarId, Done);
  if nTermExpr ≠ nil then dispose(nTermExpr, Done);
  end;
```

This code is used in section 901.

905. Existential elimination. We continue plugging along with the statements, and existential elimination (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 ";" .
```

906. ⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡

```
ChoiceStatementPtr = ↑ChoiceStatementObj;
ChoiceStatementObj = object (MObject)
  nQualVars: PList;
  nConditions: PList;
  nJustification: SimpleJustificationPtr;
  constructor Init(aQualVars, aConds : PList; aJustification : SimpleJustificationPtr);
  destructor Done; virtual;
end ;
```

907. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *ChoiceStatementObj.Init*(*aQualVars*, *aConds* : *PList*; *aJustification* : *SimpleJustificationPtr*);
  begin *nQualVars* ← *aQualVars*; *nConditions* ← *aConds*; *nJustification* ← *aJustification*;
  end;
destructor *ChoiceStatementObj.Done*;
  begin *dispose*(*nQualVars*, *Done*); *dispose*(*nConditions*, *Done*); *dispose*(*nJustification*, *Done*);
  end;

908. Conclusion statements. We recall the grammar for conclusion statements:

```
Conclusion = ( "thus" | "hence" ) ( Compact-Statement | Iterative-Equality )
  | Diffuse-Conclusion .
Diffuse-Conclusion = "thus" Diffuse-Statement | "hereby" Reasoning "end" ";" .
Iterative-Equality =
[ Label-Identifier ":" ] Term-Expression "=" Term-ExpressionSimple-Justification
                                         ".=" Term-Expression Simple-Justification
                                       { ".=" Term-Expression Simple-Justification } ";" .
```

NOTE: the whitespace in the `Iterative-Equality` rule is unimportant, but that is how Mizar users often structure them (to align the equals sign).

909. Abstract base class.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  RegularStatementKind = (*stDiffuseStatement*, *stCompactStatement*, *stIterativeEquality*);
  RegularStatementPtr = ↑*RegularStatementObj*;
  RegularStatementObj = **object** (*MObject*)
    nStatementSort: *RegularStatementKind*;
    nLab: *LabelPtr*;
    constructor *Init*(*aStatementSort* : *RegularStatementKind*);
    destructor *Done*; *virtual*;
  end ;

910. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *RegularStatementObj.Init*(*aStatementSort* : *RegularStatementKind*);
  begin *nStatementSort* ← *aStatementSort*;
  end;
destructor *RegularStatementObj.Done*;
  begin *inherited Done*;
  end;

911. Thus statement. The conclusion of a proof (idiomatically "`thus thesis`") is always a "`thus`", which Mizar calls a "diffuse statement".

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  DiffuseStatementPtr = ↑*DiffuseStatementObj*;
  DiffuseStatementObj = **object** (*RegularStatementObj*)
    constructor *Init*(*aLab* : *LabelPtr*; *aStatementSort* : *RegularStatementKind*);
    destructor *Done*; *virtual*;
  end ;

912. Constructor.

⟨ Implementation for wsmarticle.pas 854 ⟩ +≡
constructor *DiffuseStatementObj.Init*(*aLab* : *LabelPtr*; *aStatementSort* : *RegularStatementKind*);
begin *inherited Init*(*stDiffuseStatement*); *nLab* ← *aLab*; *nStatementSort* ← *aStatementSort*;
end;
destructor *DiffuseStatementObj.Done*;
begin *dispose*(*nLab*, *Done*);
end;

913. Compact statements. We recall the syntax for a compact statement is:

```
Compact-Statement = Proposition Justification ";" .
```

⟨ Publicly declared types in wsmarticle.pas 852 ⟩ +≡
CompactStatementPtr = ↑*CompactStatementObj*;
CompactStatementObj = **object** (*RegularStatementObj*)
nProp: *PropositionPtr*;
nJustification: *JustificationPtr*;
constructor *Init*(*aProp* : *PropositionPtr*; *aJustification* : *JustificationPtr*);
destructor *Done*; *virtual*;
end ;

914. Constructor.

⟨ Implementation for wsmarticle.pas 854 ⟩ +≡
constructor *CompactStatementObj.Init*(*aProp* : *PropositionPtr*; *aJustification* : *JustificationPtr*);
begin *inherited Init*(*stCompactStatement*); *nProp* ← *aProp*; *nJustification* ← *aJustification*;
end;
destructor *CompactStatementObj.Done*;
begin if *nJustification* ≠ **nil then** *dispose*(*nJustification*, *Done*);
inherited Done;
end;

915. Iterative equality. Chain of equations, where we keep transforming the right-hand side until we arrive at the desired outcome.

⟨ Publicly declared types in wsmarticle.pas 852 ⟩ +≡
IterativeStepPtr = ↑*IterativeStepObj*;
IterativeStepObj = **object** (*MObject*)
nIterPos: *Position*;
nTerm: *TermPtr*;
nJustification: *SimpleJustificationPtr*;
constructor *Init*(**const** *aPos*: *Position*; *aTerm*: *TermPtr*; *aJustification*: *JustificationPtr*);
destructor *Done*; *virtual*;
end ;

916. ⟨ Publicly declared types in wsmarticle.pas 852 ⟩ +≡
IterativeEqualityPtr = ↑*IterativeEqualityObj*;
IterativeEqualityObj = **object** (*CompactStatementObj*)
nIterSteps: *PList*;
constructor *Init*(*aProp* : *PropositionPtr*; *aJustification* : *JustificationPtr*; *aIters* : *PList*);
destructor *Done*; *virtual*;
end ;

917. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

constructor *IterativeStepObj.Init*(**const** *aPos*: *Position*; *aTerm*: *TermPtr*; *aJustification*: *JustificationPtr*);
  begin *nIterPos* ← *aPos*; *nTerm* ← *aTerm*; *nJustification* ← *SimpleJustificationPtr*(*aJustification*);
  end;
destructor *IterativeStepObj.Done*;
  begin *dispose*(*nTerm*, *Done*); *dispose*(*nJustification*, *Done*);
  end;

918. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

constructor *IterativeEqualityObj.Init*(*aProp* : *PropositionPtr*; *aJustification* : *JustificationPtr*; *aIters* : *PList*);
  begin *inherited Init*(*aProp*, *aJustification*); *nStatementSort* ← *stIterativeEquality*; *nIterSteps* ← *aIters*;
  end;
destructor *IterativeEqualityObj.Done*;
  begin *dispose*(*nIterSteps*, *Done*); *inherited Done*;
  end;

919. Remaining proof steps? So where are the other proof steps like `let` or `assume`? Well, these are handled as "generic text items" and use the *TextItem* class (§860).

Section 21.6. STRUCTURES

920. Just an aside first on "what is a structure in Mathematics?" Logic textbooks assume an *intuitive* (i.e., not formal) "finitary metatheory" following Hilbert and his famous Programme in the foundations of Mathematics. We will build a "skyscraper" atop this foundation of finitary metatheory. The first thing we do is describe a logic, the first floor in our sky scraper. This "Logic #1" is the metalogic we use to construct an axiomatic set theory, "Set Theory #2". We use "Set Theory #2" to construct another floor, a "Logic #3", which then builds another floor "Set Theory #4", and so on. We can potentially iterate building as many floors as we want, but 4 is sufficient for our purposes.

We **assert** that "Set Theory #2" *is* the Platonic "mathematical reality". Then "Logic #3" is the (ambient) logic we use to do Mathematics; it is purely "syntactic", a language for expressing proofs and definitions. Mizar's proof steps, formulas, and definitions corresponds to "Logic #3". With it, we describe an axiomatic "Set Theory #4", which is Tarski–Grothendieck set theory for Mizar. Sketching this situation out diagrammatically:

Set Theory #4	(Where we work) [syntactic]
Logic #3	"Object logic" (Where we write proofs) [syntactic]
Set Theory #2	"Mathematical Reality" [semantic]
Logic #1	"Metalogic"
Finitary Metatheory	

Fig. 5. Mathematical Platonism as a skyscaper.

Now, "mathematical objects" live in "Set Theory #2". Model theory studies structures (objects in "Set Theory #2") of theories (described in "Logic #3"). Since we "believe" that set theory "describes reality", that means we just need to describe ["syntactic"] theories using "Set Theory #4" and their "real world occurrences" in "Set Theory #2". (Well, this is a gloss, model theory sets up two additional floors in the skyscraper, and studies "models" of theories described using Logic #5 and Set Theory #6 in Set Theory #4 — and we pretend it describes the relationship between Set Theory #2 and the "syntactic floors" of the Mathematical skyscraper.)

How do we *syntactically* describe these "structures"? Well, we *know* they are not "first-class citizens" in Mizar, in the sense that they are not "just" a tuple. How do we know this? Gilbert Lee and Piotr Rudnicki's "Alternative Aggregates in Mizar" (in *MKM 2007*, Springer, pp.327–341; `doi:10.1007/978-3-540-73086-6_26`) discuss how to implement first-class structures in Mizar. This means that *technically* structures live in Logic #3. Field symbols are terms in Logic #3.

921. Why do we need this convoluted skyscraper? Without it, how do we describe a "true" formula? We can only speak of a *provable* formula. Bourbaki's *Theory of Sets* (I §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.

922. Aside: finitary metatheory, programming languages, implementing proof assistants. How does that diagram in Figure 5 of the last section compare to the *actual implementation* of Mizar? Well, a proof assistant replaces the "finitary metatheory" with an actual programming language. Then, since only Mathematical Platonists care about the "Metalogic" and "Mathematical reality", we jump ahead to implement Logic #3 — this is what happens in Mizar and other proof assistants: we implement a "purely formal" (purely syntactic) logic using a programming language. Curiously, this reflects Bourbaki's approach to the foundations of Mathematics.

We should note that programming languages are strictly stronger than finitary metatheory, since programming languages are *Turing complete*. This means they support general recursion, whereas finitary metatheory supports only primitive recursive functions. For an example of a "programming language" which is equally as strong as a finitary metatheory, see Albert R. Meyer and Dennis M. Ritchie, "The complexity of loop programs" (*ACM '67 Proc.*, 1967, `doi:10.1145/800196.806014`).

Is Turing completeness "too much" for a finitary metatheory? The short answer is: yes. Even restricting a Turing complete programming language is "too much" to be finitary. Gödel's System T was developed to preserve the "constructive character" while jettisoning the "finitary character" of Hilbert's finitary metatheory, and System T is not even Turing complete. See Kurt Gödel's *Collected Works* (vol. II, Oxford University Press, `doi:10.1093/oso/9780195147216.001.0001`, 1989; viz., pp. 245–247) for his discussion of System T. The interested reader should consult David A. Turner's "Elementary strong functional programming" (in *Int. Symp. on Funct. Program. Lang. in Educ.*, eds P.H. Hartel and R. Plasmeijer, Springer, pages 1–13, `doi:10.1007/3-540-60675-0_35`) for how to obtain System T by restricting any statically typed functional programming language.

923. Grammar for structures. We can recall the syntax for structures and fields:

```
Structure-Definition =
  "struct" [ "(" Ancestors ")" ] Structure-Symbol [ "over" Loci ] "(#" Fields "#)" ";" .
Ancestors = Structure-Type-Expression { "," Structure-Type-Expression } .
Structure-Symbol = Symbol .
Loci = Locus { "," Locus } .
Fields = Field-Segment { "," Field-Segment } .
Locus = Variable-Identifier .
Variable-Identifier = Identifier .
Field-Segment = Selector-Symbol { "," Selector-Symbol } Specification .
Selector-Symbol = Symbol .
```

924. Field symbol. A "field symbol" refers to the identifier used for a field in a structure, but not its type.

⟨Publicly declared types in `wsmarticle.pas` 852⟩ +≡
  FieldSymbolPtr = ↑*FieldSymbolObj*;
  FieldSymbolObj = **object** (*MObject*)
    nFieldPos: *Position*;
    nFieldSymbol: *integer*;
    constructor *Init*(**const** *aPos*: *Position*; *aFieldSymbNr*: *integer*);
  end ;

925. Constructor.

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
constructor *FieldSymbolObj*.*Init*(**const** *aPos*: *Position*; *aFieldSymbNr*: *integer*);
  begin *nFieldPos* ← *aPos*; *nFieldSymbol* ← *aFieldSymbNr*;
  end;

926. Field segment. A field segment refers to a list of 1 or more field symbols, and the associated type it has.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  FieldSegmentPtr = ↑*FieldSegmentObj*;
  FieldSegmentObj = **object** (*MObject*)
    nFieldSegmPos: *Position*;
    nFields: *PList*;
    nSpecification: *TypePtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aFields*: *PList*; *aSpec*: *TypePtr*);
    destructor *Done*; *virtual*;
  end ;

927. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *FieldSegmentObj.Init*(**const** *aPos*: *Position*; *aFields*: *PList*; *aSpec*: *TypePtr*);
  begin *nFieldSegmPos* ← *aPos*; *nFields* ← *aFields*; *nSpecification* ← *aSpec*;
  end;
destructor *FieldSegmentObj.Done*;
  begin *dispose*(*nFields*, *Done*); *dispose*(*nSpecification*, *Done*);
  end;

928. Locus. A "locus" refers to a term or type parametrizing a definition.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  LocusPtr = ↑*LocusObj*;
  LocusObj = **object** (*MObject*)
    nVarId: *integer*;
    nVarIdPos: *Position*;
    constructor *Init*(**const** *aPos*: *Position*; *aIdentNr*: *integer*);
  end ;

929. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *LocusObj.Init*(**const** *aPos*: *Position*; *aIdentNr*: *integer*);
  begin *nVarId* ← *aIdentNr*; *nVarIdPos* ← *aPos*;
  end;

930. Structure definition. Finally, structures are finite maps from selectors to terms, with structure inheritance thrown into the mix. They may be defined "`over`" a finite list of types (e.g., a module structure is "`over`" a ring). Note that we need to first introduce "patterns" before describing the structure definition, since "patterns" are needed in definitions.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  ⟨ Pattern objects (`wsmarticle.pas`) 933 ⟩
  StructureDefinitionPtr = ↑*StructureDefinitionObj*;
  StructureDefinitionObj = **object** (*MObject*)
    nStrPos: *Position*;
    nAncestors: *PList*;
    nDefStructPattern: *ModePatternPtr*;
    nSgmFields: *PList*;
    constructor *Init*(**const** *aPos*: *Position*; *aAncestors*: *PList*; *aStructSymb*: *integer*;
      aOverArgs: *PList*; *aFields*: *PList*);
    destructor *Done*; *virtual*;
  end ;

931. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *StructureDefinitionObj*.*Init*(**const** *aPos*: *Position*; *aAncestors*: *PList*;
    aStructSymb: *integer*; *aOverArgs*: *PList*; *aFields*: *PList*);
  begin *nStrPos* ← *aPos*; *nAncestors* ← *aAncestors*;
  nDefStructPattern ← *new*(*ModePatternPtr*, *Init*(*aPos*, *aStructSymb*, *aOverArgs*));
  nDefStructPattern↑.*nPatternSort* ← *itDefStruct*; *nSgmFields* ← *aFields*;
  end;
destructor *StructureDefinitionObj*.*Done*;
  begin *dispose*(*nAncestors*, *Done*); *dispose*(*nDefStructPattern*, *Done*); *dispose*(*nSgmFields*, *Done*);
  end;

Section 21.7. PATTERNS

932. A "*Pattern*" in Mizar is a format with the type information for all the arguments around a term. The notion of a "*Pattern*" also refers to the definiendum of a definition. The syntax of patterns

```
Mode-Pattern = Mode-Symbol [ "of" Loci ] .
Attribute-Pattern = Locus "is" [ Attribute-Loci ] Attribute-Symbol .
Attribute-Loci = Loci | "(" Loci ")" .
Predicate-Pattern = [ Loci ] Predicate-Symbol [ Loci ] .
Functor-Pattern = [ Functor-Loci ] Functor-Symbol [ Functor-Loci ]
  | Left-Functor-Bracket Loci Right-Functor-Bracket .
Functor-Loci = Locus | "(" Loci ")" .
```

933. Base class for patterns.

⟨ Pattern objects (`wsmarticle.pas`) 933 ⟩ ≡
  PatternPtr = ↑*PatternObj*;
  PatternObj = **object** (*mObject*)
    nPatternPos: *Position*;
    nPatternSort: *ItemKind*;
    constructor *Init*(**const** *aPos*: *Position*; *aSort*: *ItemKind*);
  end ;

See also sections 935, 937, 939, and 941.

This code is used in section 930.

934. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *PatternObj.Init*(**const** *aPos*: *Position*; *aSort*: *ItemKind*);
  begin *nPatternPos* ← *aPos*; *nPatternSort* ← *aSort*;
  end;

935. Mode patterns. The syntax for "mode patterns" looks like:

```
Mode-Pattern = Mode-Symbol [ "of" Loci ] .
```

⟨ Pattern objects (`wsmarticle.pas`) 933 ⟩ +≡
  ModePatternPtr = ↑*ModePatternObj*;
  ModePatternObj = **object** (*PatternObj*)
    nModeSymbol: *Integer*;
    nArgs: *PList*;
    constructor *Init*(**const** *aPos*: *Position*; *aSymb*: *integer*; *aArgs*: *PList*);
    destructor *Done*; *virtual*;
  end ;

936. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *ModePatternObj.Init*(**const** *aPos*: *Position*; *aSymb*: *integer*; *aArgs*: *PList*);
  begin *inherited Init*(*aPos*, *itDefMode*); *nModeSymbol* ← *aSymb*; *nArgs* ← *aArgs*;
  end;
destructor *ModePatternObj.Done*;
  begin *dispose*(*nArgs*, *Done*);
  end;

937. Attribute patterns. Attributes can have loci prefixing the attribute symbol, but *not* suffixing the attribute symbol.

```
Attribute-Pattern = Locus "is" [ Attribute-Loci ] Attribute-Symbol .
Attribute-Loci = Loci | "(" Loci ")" .
```

⟨ Pattern objects (`wsmarticle.pas`) 933 ⟩ +≡
  AttributePatternPtr = ↑*AttributePatternObj*;
  AttributePatternObj = **object** (*PatternObj*)
    nAttrSymbol: *Integer*;
    nArg: *LocusPtr*;
    nArgs: *PList*;
    constructor *Init*(**const** *aPos*: *Position*; *aArg*: *LocusPtr*; *aSymb*: *integer*; *aArgs*: *PList*);
    destructor *Done*; *virtual*;
  end ;

938. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *AttributePatternObj.Init*(**const** *aPos*: *Position*; *aArg*: *LocusPtr*; *aSymb*: *integer*; *aArgs*: *PList*);
  begin *inherited Init*(*aPos*, *itDefAttr*); *nAttrSymbol* ← *aSymb*; *nArg* ← *aArg*; *nArgs* ← *aArgs*;
  end;
destructor *AttributePatternObj.Done*;
  begin *dispose*(*nArg*, *Done*); *dispose*(*nArgs*, *Done*);
  end;

939. Predicate patterns. Predicates can have loci on either side of the predicate symbol, without requiring parentheses (unlike functors).

```
Predicate-Pattern = [ Loci ] Predicate-Symbol [ Loci ] .
```

⟨ Pattern objects (`wsmarticle.pas`) 933 ⟩ +≡
  PredicatePatternPtr = ↑*PredicatePatternObj*;
  PredicatePatternObj = **object** (*PatternObj*)
    nPredSymbol: *Integer*;
    nLeftArgs, *nRightArgs*: *PList*;
    constructor *Init*(**const** *aPos*: *Position*; *aLArgs*: *PList*; *aSymb*: *integer*; *aRArgs*: *PList*);
    destructor *Done*; *virtual*;
  end ;

940. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *PredicatePatternObj.Init*(**const** *aPos*: *Position*;
    aLArgs: *PList*; *aSymb*: *integer*; *aRArgs*: *PList*);
  begin *inherited Init*(*aPos*, *itDefPred*); *nPredSymbol* ← *aSymb*; *nLeftArgs* ← *aLArgs*;
  nRightArgs ← *aRArgs*;
  end;
destructor *PredicatePatternObj.Done*;
  begin *dispose*(*nLeftArgs*, *Done*); *dispose*(*nRightArgs*, *Done*);
  end;

941. Functor pattern. Functors can have loci on either side. If more than one locus is used on one side, then it must be placed in parentheses and comma-separated. The syntax:

```
Functor-Pattern = [ Functor-Loci ] Functor-Symbol [ Functor-Loci ]
  | Left-Functor-Bracket Loci Right-Functor-Bracket .
Functor-Loci = Locus | "(" Loci ")" .
```

⟨ Pattern objects (`wsmarticle.pas`) 933 ⟩ +≡
FunctorSort = (*InfixFunctor*, *CircumfixFunctor*);
FunctorPatternPtr = ↑*FunctorPatternObj*;
FunctorPatternObj = **object** (*PatternObj*)
nFunctKind: *FunctorSort*;
constructor *Init*(**const** *aPos*: *Position*; *aKind*: *FunctorSort*);
end ;
CircumfixFunctorPatternPtr = ↑*CircumfixFunctorPatternObj*;
CircumfixFunctorPatternObj = **object** (*FunctorPatternObj*)
nLeftBracketSymb, *nRightBracketSymb*: *integer*;
nArgs: *PList*;
constructor *Init*(**const** *aPos*: *Position*; *aLBSymb*, *aRBSymb*: *integer*; *aArgs*: *PList*);
destructor *Done*; *virtual*;
end ;
InfixFunctorPatternPtr = ↑*InfixFunctorPatternObj*;
InfixFunctorPatternObj = **object** (*FunctorPatternObj*)
nOperSymb: *integer*;
nLeftArgs, *nRightArgs*: *PList*;
constructor *Init*(**const** *aPos*: *Position*; *aLArgs*: *PList*; *aSymb*: *integer*; *aRArgs*: *PList*);
destructor *Done*; *virtual*;
end ;

942. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *FunctorPatternObj*.*Init*(**const** *aPos*: *Position*; *aKind*: *FunctorSort*);
begin *inherited Init*(*aPos*, *itDefFunc*); *nFunctKind* ← *aKind*;
end;
constructor *CircumfixFunctorPatternObj*.*Init*(**const** *aPos*: *Position*; *aLBSymb*, *aRBSymb*: *integer*; *aArgs*: *PList*);
begin *inherited Init*(*aPos*, *CircumfixFunctor*); *nLeftBracketSymb* ← *aLBSymb*;
nRightBracketSymb ← *aRBSymb*; *nArgs* ← *aArgs*;
end;
destructor *CircumfixFunctorPatternObj*.*Done*;
begin *dispose*(*nArgs*, *Done*);
end;
constructor *InfixFunctorPatternObj*.*Init*(**const** *aPos*: *Position*; *aLArgs*: *PList*; *aSymb*: *integer*; *aRArgs*: *PList*);
begin *inherited Init*(*aPos*, *InfixFunctor*); *nOperSymb* ← *aSymb*; *nLeftArgs* ← *aLArgs*;
nRightArgs ← *aRArgs*;
end;
destructor *InfixFunctorPatternObj*.*Done*;
begin *dispose*(*nLeftArgs*, *Done*); *dispose*(*nRightArgs*, *Done*);
end;

Section 21.8. DEFINITIONS

943. In Mizar, we can redefine an existing definition (either changing the type of a term or "the right hand side" of a definition) *or* we can introduce a new definition. There are 5 different things we can introduce: structures, modes [types], functors [terms], predicates, and attributes. Rather than bombard the reader with a long chunk of grammar, let us divide it up into easy-to-digest pieces. The basic block structure of a definition is the same for all these situations, its grammar looks like:

```
Definitional-Item = Definitional-Block ";" .
Definitional-Block = "definition" { Definition-Item | Definition | Redefinition } "end" .
Definition-Item = Loci-Declaration | Permissive-Assumption | Auxiliary-Item .
Loci-Declaration = "let" Qualified-Variables [ "such" Conditions ] ";" .
Permissive-Assumption = Assumption .
Definition = Structure-Definition
  | Mode-Definition
  | Functor-Definition
  | Predicate-Definition
  | Attribute-Definition .
```

944. Redefinitions. Redefinitions allow us to alter the type or meaning of a definition. This isn't willy-nilly, the user still needs to prove the redefined version is logically equivalent to the initial definition.

```
Redefinition =
  "redefine" ( Mode-Definition | Functor-Definition | Predicate-Definition | Attribute-Definition ) .
```

945. Structure definitions. Structures intuitively correspond to new "gadgets" (sets equipped with extra structure), which is often presented in Mathematics as "just another tuple". Mizar allows structures to inherit other structures, so a topological group extends a topological space structure *and* a magma structure (since a group in Mizar is a magma with some extra properties).

```
Structure-Definition =
  "struct" [ "(" Ancestors ")" ] Structure-Symbol [ "over" Loci ] "(#" Fields "#)" ";" .
Ancestors = Structure-Type-Expression { "," Structure-Type-Expression } .
Structure-Symbol = Symbol .
Loci = Locus { "," Locus } .
Fields = Field-Segment { "," Field-Segment } .
Locus = Variable-Identifier .
Variable-Identifier = Identifier .
Field-Segment = Selector-Symbol { "," Selector-Symbol } Specification .
Selector-Symbol = Symbol .
Specification = "->" Type-Expression .
```

946. Definiens. Recall the grammar for `Definiens` looks like:

```
Definiens = Simple-Definiens | Conditional-Definiens .
Simple-Definiens = [ ":" Label-Identifier ":" ] ( Sentence | Term-Expression ) .
Label-Identifier = Identifier .
Conditional-Definiens = [ ":" Label-Identifier ":" ] Partial-Definiens-List
  [ "otherwise" ( Sentence | Term-Expression ) ] .
Partial-Definiens-List = Partial-Definiens { "," Partial-Definiens } .
Partial-Definiens = ( Sentence | Term-Expression ) "if" Sentence .
```

We begin with a base class for definiens. This is extended by *SimpleDefiniens* and *ConditionalDefiniens* classes.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  HowToDefine = (*dfEmpty*, *dfMeans*, *dfEquals*);
  DefiniensSort = (*SimpleDefiniens*, *ConditionalDefiniens*);
  DefiniensPtr = ↑*DefiniensObj*;
  DefiniensObj = **object** (*MObject*)
    nDefSort: *DefiniensSort*;
    nDefPos: *Position*;
    nDefLabel: *LabelPtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aLab*: *LabelPtr*; *aKind*: *DefiniensSort*);
    destructor *Done*; *virtual*;
  end ;

947. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *DefiniensObj.Init*(**const** *aPos*: *Position*; *aLab*: *LabelPtr*; *aKind*: *DefiniensSort*);
  begin *nDefSort* ← *aKind*; *nDefPos* ← *aPos*; *nDefLabel* ← *aLab*;
  end;
destructor *DefiniensObj.Done*;
  begin if *nDefLabel* ≠ **nil then** *dispose*(*nDefLabel*, *Done*);
  end;

948. Definiens expression. These nodes in the abstract syntax tree describe "the right hand side" of a definition. A simple definiens is just a pointer to one definiens expression object, for example.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  DefExpressionPtr = ↑*DefExpressionObj*;
  DefExpressionObj = **object** (*MObject*)
    nExprKind: *ExpKind*;
    nExpr: *PObject*;
    constructor *Init*(*aKind* : *ExpKind*; *aExpr* : *PObject*);
    destructor *Done*; *virtual*;
  end ;

949. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *DefExpressionObj.Init*(*aKind* : *ExpKind*; *aExpr* : *Pobject*);
  begin *nExprKind* ← *aKind*; *nExpr* ← *aExpr*;
  end;
destructor *DefExpressionObj.Done*;
  begin *dispose*(*nExpr*, *Done*);
  end;

950. Simple definiens. This is the "default" definiens, i.e., the definiens which are not "by cases".

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  SimpleDefiniensPtr = ↑*SimpleDefiniensObj*;
  SimpleDefiniensObj = **object** (*DefiniensObj*)
    nExpression: *DefExpressionPtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aLab*: *LabelPtr*; *aDef*: *DefExpressionPtr*);
    destructor *Done*; *virtual*;
  end ;

951. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *SimpleDefiniensObj.Init*(**const** *aPos*: *Position*; *aLab*: *LabelPtr*; *aDef*: *DefExpressionPtr*);
  begin *inherited Init*(*aPos*, *aLab*, *SimpleDefiniens*); *nExpression* ← *aDef*;
  end;
destructor *SimpleDefiniensObj.Done*;
  begin *dispose*(*nExpression*, *Done*); *inherited Done*;
  end;

952. Definition for particular case. We have "⟨*sentence or term*⟩ `if` ⟨*guard condition*⟩" represented by a couple of pointers: one to the "sentence or term" definiens, and the second to the "guard" condition.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  PartDefPtr = ↑*PartDefObj*;
  PartDefObj = **object** (*MObject*)
    nPartDefiniens: *DefExpressionPtr*;
    nGuard: *FormulaPtr*;
    constructor *Init*(*aPartDef* : *DefExpressionPtr*; *aGuard* : *FormulaPtr*);
    destructor *Done*; *virtual*;
  end ;

953. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *PartDefObj.Init*(*aPartDef* : *DefExpressionPtr*; *aGuard* : *FormulaPtr*);
  begin *nGuard* ← *aGuard*; *nPartDefiniens* ← *aPartDef*;
  end;
destructor *PartDefObj.Done*;
  begin *dispose*(*nPartDefiniens*, *Done*); *dispose*(*nGuard*, *Done*);
  end;

954. Conditional definiens. A conditional definiens consists of a finite list of pointers to *PartDef* objects, and a pointer to the default "`otherwise`" definien.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  ConditionalDefiniensPtr = ↑*ConditionalDefiniensObj*;
  ConditionalDefiniensObj = **object** (*DefiniensObj*)
    nConditionalDefiniensList: *PList*;
    nOtherwise: *DefExpressionPtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aLab*: *LabelPtr*; *aPartialDefs*: *PList*;
      aOtherwise: *DefExpressionPtr*);
    destructor *Done*; *virtual*;
  end ;

955. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

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;

956. Mode definitions. Mizar was heavily inspired by ALGOL, and even borrows ALGOL's terminology for types ("modes"). These are "soft types", which are predicates in the ambient logic.

However, we need to establish the well-definedness of types (i.e., they are inhabited by at least one term), or else we end up in "free logic". For example, if `EmptyType` is a hypothetical empty type, then `for x being EmptyType holds P[x]` is always true, and `ex x being EmptyType st P[x]` is always false. The clever Mizar user can abuse this, and end up compromising the soundness of classical logic. To avert catastrophe, we require proving there exists at least one term of the newly defined type.

```
Mode-Definition = "mode" Mode-Pattern
  ( [ Specification ] [ "means" Definiens ] ";" Correctness-Conditions | "is" Type-Expression ";" )
  { Mode-Property } .
Mode-Pattern = Mode-Symbol [ "of" Loci ] .
Mode-Symbol = Symbol | "set" .
Mode-Synonym = "synonym" Mode-Pattern "for" Mode-Pattern ";" .
Mode-Property = "sethood" Justification ";" .
```

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
 ModeDefinitionSort = (*defExpandableMode*, *defStandardMode*);
 ModeDefinitionPtr = ↑*ModeDefinitionObj*;
 ModeDefinitionObj = **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 ;

957. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

constructor *ModeDefinitionObj.Init*(**const** *aPos*: *Position*; *aDefKind*: *ModeDefinitionSort*;
  aRedef: *boolean*; *aPattern*: *ModePatternPtr*);
 begin *nDefKind* ← *aDefKind*; *nDefModePos* ← *aPos*; *nRedefinition* ← *aRedef*;
 nDefModePattern ← *aPattern*;
 end;
destructor *ModeDefinitionObj.Done*;
 begin *dispose*(*nDefModePattern*, *Done*);
 end;

958. Expandable mode definitions. These are simple "abbreviations" of modes which are of the form "`mode` ⟨*type name*⟩ `is` ⟨$adjective_1$⟩ ⋯ ⟨$adjective_n$⟩ ⟨*type*⟩", i.e., just a stack of adjectives atop a type.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  ExpandableModeDefinitionPtr = ↑*ExpandableModeDefinitionObj*;
  ExpandableModeDefinitionObj = **object** (*ModeDefinitionObj*)
    nExpansion: *TypePtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aPattern*: *ModePatternPtr*; *aExp*: *TypePtr*);
    destructor *Done*; *virtual*;
  end ;

959. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *ExpandableModeDefinitionObj.Init*(**const** *aPos*: *Position*;
    aPattern: *ModePatternPtr*; *aExp*: *TypePtr*);
  begin *inherited Init*(*aPos*, *defExpandableMode*, *false*, *aPattern*); *nExpansion* ← *aExp*;
  end;
destructor *ExpandableModeDefinitionObj.Done*;
  begin *dispose*(*nExpansion*, *Done*); *inherited Done*;
  end;

960. Standard mode definitions.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  StandardModeDefinitionPtr = ↑*StandardModeDefinitionObj*;
  StandardModeDefinitionObj = **object** (*ModeDefinitionObj*)
    nSpecification: *TypePtr*;
    nDefiniens: *DefiniensPtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aRedef*: *boolean*; *aPattern*: *ModePatternPtr*;
      aSpec: *TypePtr*; *aDef*: *DefiniensPtr*);
    destructor *Done*; *virtual*;
  end ;

961. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *StandardModeDefinitionObj.Init*(**const** *aPos*: *Position*;
    aRedef: *boolean*; *aPattern*: *ModePatternPtr*; *aSpec*: *TypePtr*; *aDef*: *DefiniensPtr*);
  begin *inherited Init*(*aPos*, *defStandardMode*, *aRedef*, *aPattern*); *nSpecification* ← *aSpec*;
  nDefiniens ← *aDef*;
  end;
destructor *StandardModeDefinitionObj.Done*;
  begin *dispose*(*nSpecification*, *Done*); *dispose*(*nDefiniens*, *Done*); *inherited Done*;
  end;

962. Attribute definitions. Attributes, like predicates, do not need to worry about correctness conditions. It's only when we want to use them like adjectives on a type that we need to worry, but that's a `registration` block concern.

```
Attribute-Definition = "attr" Attribute-Pattern "means" Definiens ";" Correctness-Conditions .
Attribute-Pattern = Locus "is" [ Attribute-Loci ] Attribute-Symbol .
```

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
AttributeDefinitionPtr = ↑*AttributeDefinitionObj*;
AttributeDefinitionObj = **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 ;

963. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *AttributeDefinitionObj.Init*(**const** *aPos*: *Position*;
aRedef: *boolean*; *aPattern*: *AttributePatternPtr*; *aDef*: *DefiniensPtr*);
begin *nDefAttrPos* ← *aPos*; *nRedefinition* ← *aRedef*; *nDefAttrPattern* ← *aPattern*;
nDefiniens ← *aDef*;
end;

destructor *AttributeDefinitionObj.Done*;
begin *dispose*(*nDefAttrPattern*, *Done*); *dispose*(*nDefiniens*, *Done*);
end;

964. Predicate definitions. Predicates are among the less demanding of the definitions: they are always well-defined, so we do not need to worry about correctness conditions.

```
Predicate-Definition = "pred" Predicate-Pattern [ "means" Definiens ] ";"
Correctness-Conditions { Predicate-Property } .
Predicate-Pattern = [ Loci ] Predicate-Symbol [ Loci ] .
Predicate-Property = ("symmetry" | "asymmetry" | "connectedness" | "reflexivity" | "irreflexivity")
  Justification ";" .
Predicate-Synonym = "synonym" Predicate-Pattern "for" Predicate-Pattern ";" .
Predicate-Antonym = "antonym" Predicate-Pattern "for" Predicate-Pattern ";" .
Predicate-Symbol = Symbol | "=" .
```

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
PredicateDefinitionPtr = ↑*PredicateDefinitionObj*;
PredicateDefinitionObj = **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 ;

965. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
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;

966. Functor definitions. We can also define new terms. Well, they introduce "term constructors" (constructors for terms). Mizar calls these guys "functors".

Functor definitions need to establish the well-definedness of the new term constructor. What this means depends on whether we define the new term using "means" or "equals", i.e.,

(1) "⟨*new term*⟩ `means` ⟨*formula*⟩" requires proving the existence and uniqueness of the new term;
(2) "⟨*new term*⟩ `equals` ⟨*term 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 | "}" | "]" .
```

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  FunctorDefinitionPtr = ↑*FunctorDefinitionObj*;
  FunctorDefinitionObj = **object** (*MObject*)
    nDefFuncPos: *Position*;
    nDefFuncPattern: *FunctorPatternPtr*;
    nRedefinition: *boolean*;
    nSpecification: *TypePtr*;
    nDefiningWay: *HowToDefine*;
    nDefiniens: *DefiniensPtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aRedef*: *boolean*; *aPattern*: *FunctorPatternPtr*;
      aSpec: *TypePtr*; *aDefWay*: *HowToDefine*; *aDef*: *DefiniensPtr*);
    destructor *Done*; *virtual*;
  end ;

967. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *FunctorDefinitionObj.Init*(**const** *aPos*: *Position*; *aRedef*: *boolean*;
    aPattern: *FunctorPatternPtr*; *aSpec*: *TypePtr*; *aDefWay*: *HowToDefine*; *aDef*: *DefiniensPtr*);
  begin *nDefFuncPos* ← *aPos*; *nRedefinition* ← *aRedef*; *nDefFuncPattern* ← *aPattern*;
  nSpecification ← *aSpec*; *nDefiningWay* ← *aDefWay*; *nDefiniens* ← *aDef*;
  end;
destructor *FunctorDefinitionObj.Done*;
  begin *dispose*(*nDefFuncPattern*, *Done*); *dispose*(*nDefiniens*, *Done*);
  end;

968. Notation block. We can recall the syntax for notation blocks.

```
Notation-Block = "notation" { Loci-Declaration | Notation-Declaration } "end" .
Notation-Declaration = Mode-Synonym
  | Functor-Synonym
  | Attribute-Synonym | Attribute-Antonym
  | Predicate-Synonym | Predicate-Antonym .
Mode-Synonym = "synonym" Mode-Pattern "for" Mode-Pattern ";" .
Functor-Synonym = "synonym" Functor-Pattern "for" Functor-Pattern ";" .
Predicate-Synonym = "synonym" Predicate-Pattern "for" Predicate-Pattern ";" .
Predicate-Antonym = "antonym" Predicate-Pattern "for" Predicate-Pattern ";" .
Attribute-Synonym = "synonym" Attribute-Pattern "for" Attribute-Pattern ";" .
Attribute-Antonym = "antonym" Attribute-Pattern "for" Attribute-Pattern ";" .
```

The reader will observe all these notation items relate a new pattern which is either a synonym or antonym for an old pattern. That is to say, we only need two patterns to store as data in a notation item node in the abstract syntax tree.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  NotationDeclarationPtr = ↑*NotationDeclarationObj*;
  NotationDeclarationObj = **object** (*mObject*)
    nNotationPos: *Position*;
    nNotationSort: *ItemKind*;
    nOriginPattern, *nNewPattern*: *PatternPtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aNSort*: *ItemKind*; *aNewPatt*, *aOrigPatt*: *PatternPtr*);
    destructor *Done*; *virtual*;
  end ;

969. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *NotationDeclarationObj.Init*(**const** *aPos*: *Position*; *aNSort*: *ItemKind*;
    aNewPatt, *aOrigPatt*: *PatternPtr*);
  begin *nNotationPos* ← *aPos*; *nNotationSort* ← *aNSort*; *nOriginPattern* ← *aOrigPatt*;
  nNewPattern ← *aNewPatt*;
  end;
destructor *NotationDeclarationObj.Done*;
  begin *dispose*(*nOriginPattern*, *Done*); *dispose*(*nNewPattern*, *Done*);
  end;

970. Assumptions in a definition block. The syntax for assumptions in a definition block looks like:

```
Assumption = Single-Assumption | Collective-Assumption | Existential-Assumption .
Single-Assumption = "assume" Proposition ";" .
Collective-Assumption = "assume" Conditions ";" .
Existential-Assumption = "given" Qualified-Variables [ "such" Conditions ] ";" .
Conditions = "that" Proposition { "and" Proposition } .
Proposition = [ Label-Identifier ":" ] Sentence .
Sentence = Formula-Expression .
```

⟨Publicly declared types in `wsmarticle.pas` 852⟩ +≡
AssumptionKind = (*SingleAssumption*, *CollectiveAssumption*, *ExistentialAssumption*);
AssumptionPtr = ↑*AssumptionObj*;
AssumptionObj = **object** (*MObject*)
  nAssumptionPos: *Position*;
  nAssumptionSort: *AssumptionKind*;
  constructor *Init*(**const** *aPos*: *Position*; *aSort*: *AssumptionKind*);
end ;

971. Constructor.

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
constructor *AssumptionObj.Init*(**const** *aPos*: *Position*; *aSort*: *AssumptionKind*);
  begin *nAssumptionPos* ← *aPos*; *nAssumptionSort* ← *aSort*;
  end;

972. Single assumption. When a definition has a single assumption, i.e., a single (usually labeled) formula.

⟨Publicly declared types in `wsmarticle.pas` 852⟩ +≡
SingleAssumptionPtr = ↑*SingleAssumptionObj*;
SingleAssumptionObj = **object** (*AssumptionObj*)
  nProp: *PropositionPtr*;
  constructor *Init*(**const** *aPos*: *Position*; *aProp*: *PropositionPtr*);
  destructor *Done*; *virtual*;
end ;

973. Constructor.

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
constructor *SingleAssumptionObj.Init*(**const** *aPos*: *Position*; *aProp*: *PropositionPtr*);
  begin *inherited Init*(*aPos*, *SingleAssumption*); *nProp* ← *aProp*;
  end;
destructor *SingleAssumptionObj.Done*;
  begin *dispose*(*nProp*, *Done*);
  end;

974. Collective assumption. This describes the case when the assumption is "`assume` C_1 `and` ... `and` C_n".

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  CollectiveAssumptionPtr = ↑*CollectiveAssumptionObj*;
  CollectiveAssumptionObj = **object** (*AssumptionObj*)
    nConditions: *PList*;
    constructor *Init*(**const** *aPos*: *Position*; *aProps*: *PList*);
    destructor *Done*; *virtual*;
  end ;

975. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *CollectiveAssumptionObj.Init*(**const** *aPos*: *Position*; *aProps*: *PList*);
  begin *inherited Init*(*aPos*, *CollectiveAssumption*); *nConditions* ← *aProps*;
  end;
destructor *CollectiveAssumptionObj.Done*;
  begin *dispose*(*nConditions*, *Done*);
  end;

976. Existential assumption. I must confess I am surprised to see an existential assumption node being a subclass of a collective assumption node.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  ExistentialAssumptionPtr = ↑*ExistentialAssumptionObj*;
  ExistentialAssumptionObj = **object** (*CollectiveAssumptionObj*)
    nQVars: *PList*;
    constructor *Init*(**const** *aPos*: *Position*; *aQVars*, *aProps*: *PList*);
    destructor *Done*; *virtual*;
  end ;

977. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *ExistentialAssumptionObj.Init*(**const** *aPos*: *Position*; *aQVars*, *aProps*: *PList*);
  begin *AssumptionObj.Init*(*aPos*, *CollectiveAssumption*); *nConditions* ← *aProps*; *nQVars* ← *aQVars*;
  end;
destructor *ExistentialAssumptionObj.Done*;
  begin *dispose*(*nQVars*, *Done*); *inherited Done*;
  end;

978. 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.

⟨Publicly declared types in `wsmarticle.pas` 852⟩ +≡
```
  CorrectnessPtr = ↑CorrectnessObj;
  CorrectnessObj = object (MObject)
    nCorrCondPos: Position;
    nJustification: JustificationPtr;
    constructor Init(const aPos: Position; aJustification: JustificationPtr);
    destructor Done; virtual;
  end ;
```

979. Constructor.

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
```
constructor CorrectnessObj.Init(const aPos: Position; aJustification: JustificationPtr);
  begin nCorrCondPos ← aPos; nJustification ← aJustification;
  end;
destructor CorrectnessObj.Done;
  begin dispose(nJustification, Done);
  end;
```

980. Correctness condition. For the correctness condition associated with a definition or registration, we have this *CorrectnessCondition* object. When we need multiple correctness conditions, we extend it with a subclass.

⟨Publicly declared types in `wsmarticle.pas` 852⟩ +≡
```
  CorrectnessConditionPtr = ↑CorrectnessConditionObj;
  CorrectnessConditionObj = object (CorrectnessObj)
    nCorrCondSort: CorrectnessKind;
    constructor Init(const aPos: Position; aSort: CorrectnessKind; aJustification: JustificationPtr);
    destructor Done; virtual;
  end ;
```

981. Constructor.

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
```
constructor CorrectnessConditionObj.Init(const aPos: Position;
      aSort: CorrectnessKind; aJustification: JustificationPtr);
  begin inherited Init(aPos, aJustification); nCorrCondSort ← aSort;
  end;
destructor CorrectnessConditionObj.Done;
  begin inherited Done;
  end;
```

982. 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.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  CorrectnessConditionsSet = **set of** *CorrectnessKind*;
  CorrectnessConditionsPtr = ↑*CorrectnessConditionsObj*;
  CorrectnessConditionsObj = **object** (*CorrectnessObj*)
    nConditions: *CorrectnessConditionsSet*;
    constructor *Init*(**const** *aPos*: *Position*; **const** *aConds*: *CorrectnessConditionsSet*;
      aJustification: *JustificationPtr*);
    destructor *Done*; *virtual*;
  end ;

983. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *CorrectnessConditionsObj.Init*(**const** *aPos*: *Position*;
  const *aConds*: *CorrectnessConditionsSet*;
      aJustification: *JustificationPtr*);
  begin *inherited Init*(*aPos*, *aJustification*); *nConditions* ← *aConds*;
  end;
destructor *CorrectnessConditionsObj.Done*;
  begin *inherited Done*;
  end;

984. Definition properties. The grammar for properties in a definition looks like:

```
Mode-Property = "sethood" Justification ";" .
Functor-Property = ("commutativity" | "idempotence" | "involutiveness" | "projectivity")
  Justification ";" .
Predicate-Property = ("symmetry" | "asymmetry" | "connectedness" | "reflexivity" | "irreflexivity")
  Justification ";" .
```

We see these are all, more or less, "the same": we have a "kind" of property and a justification. We recall (§720) that we have already introduced the "kind" of properties. So the class describing a definition property node in the abstract syntax tree is:

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  PropertyPtr = ↑*PropertyObj*;
  PropertyObj = **object** (*MObject*)
    nPropertyPos: *Position*;
    nPropertySort: *PropertyKind*;
    nJustification: *JustificationPtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aSort*: *PropertyKind*; *aJustification*: *JustificationPtr*);
    destructor *Done*; *virtual*;
  end ;

985. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *PropertyObj.Init*(**const** *aPos*: *Position*; *aSort*: *PropertyKind*;
    aJustification: *JustificationPtr*);
  begin *nPropertyPos* ← *aPos*; *nPropertySort* ← *aSort*; *nJustification* ← *aJustification*;
  end;
destructor *PropertyObj.Done*;
  begin *inherited Done*;
  end;

Section 21.9. REGISTRATIONS

986. There are three "main" types of registrations, which are "cluster registrations" (because they all involve the "`cluster`" keyword):

(1) Existential registrations are of the form "`cluster` ⟨*attribute*⟩ `for` ⟨*type*⟩" and establishes that a given attribute can act as an adjective for the type.
(2) Conditional registrations are of the form "`cluster` ⟨$attribute_1$⟩ `->` ⟨$attribute_2$⟩ `for` ⟨*type*⟩" which tells Mizar that when ⟨$attribute_1$⟩ is established for a term, then Mizar can automatically add ⟨$attribute_2$⟩ for the term
(3) Functorial registrations are of the form "`cluster` ⟨*term*⟩ `->` ⟨*attribute*⟩ [`for` ⟨*type*⟩]" which will automatically add an attribute to a term.

We also have three lesser registrations which are still important:

(1) Sethood registrations, establishes a type can be used as a set in a Fraenkel term.
(2) Reduction registration, which allows Mizar's term rewriting module to use this rule when reasoning about things.
(3) Identification registration, which allows Mizar to identify terms of different types.

```
Cluster-Registration = Existential-Registration
  | Conditional-Registration
  | Functorial-Registration .

Existential-Registration = "cluster" Adjective-Cluster "for" Type-Expression ";"
  Correctness-Conditions .

Adjective-Cluster = { Adjective } .

Adjective = [ "non" ] [ Adjective-Arguments ] Attribute-Symbol .

Conditional-Registration = "cluster" Adjective-Cluster "->" Adjective-Cluster "for" Type-Expression ";"
  Correctness-Conditions .

Functorial-Registration = "cluster" Term-Expression "->" Adjective-Cluster [ "for" Type-Expression ] ";"
  Correctness-Conditions .

Identify-Registration = "identify" Functor-Pattern "with" Functor-Pattern
    [ "when" Locus "=" Locus { "," Locus "=" Locus } ] ";"
  Correctness-Conditions .

Property-Registration = "sethood" "of" Type-Expression Justification ";" .

Reduction-Registration = "reduce" Term-Expression "to" Term-Expression ";"
  Correctness-Conditions .
```

987. Cluster registration. We have a base class for the three types of cluster registrations.

⟨Publicly declared types in `wsmarticle.pas` 852⟩ +≡

ClusterRegistrationKind = (*ExistentialRegistration*, *ConditionalRegistration*, *FunctorialRegistration*);

ClusterPtr = ↑*ClusterObj*;
ClusterObj = **object** (*MObject*)
  nClusterPos: *Position*;
  nClusterKind: *ClusterRegistrationKind*;
  nConsequent: *PList*;
  nClusterType: *TypePtr*;
  constructor *Init*(**const** *aPos*: *Position*; *aKind*: *ClusterRegistrationKind*; *aCons*: *PList*;
    aTyp: *TypePtr*);
  destructor *Done*; *virtual*;
end ;

988. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *ClusterObj.Init*(**const** *aPos*: *Position*;
  aKind: *ClusterRegistrationKind*; *aCons*: *PList*; *aTyp*: *TypePtr*);
 begin *nClusterPos* ← *aPos*; *nClusterKind* ← *aKind*; *nConsequent* ← *aCons*; *nClusterType* ← *aTyp*;
 end;
destructor *ClusterObj.Done*;
 begin *dispose*(*nConsequent*, *Done*);
 end;

989. Existential cluster. We register the fact there always exists a term of a given type satisfying an attribute (e.g., "empty" for "set" means there always exists an empty set; registering the existential cluster "non empty" for "set" means there always exists a nonempty set). This means the attribute may henceforth be used as an adjective on the type.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
 EClusterPtr = ↑*EClusterObj*;
 EClusterObj = **object** (*ClusterObj*)
  constructor *Init*(**const** *aPos*: *Position*; *aCons*: *PList*; *aTyp*: *TypePtr*);
  destructor *Done*; *virtual*;
 end ;

990. Constructor. There are no additional fields to an existential cluster object, so it literally passes the parameters onto the superclass's constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *EClusterObj.Init*(**const** *aPos*: *Position*; *aCons*: *PList*; *aTyp*: *TypePtr*);
 begin *ClusterObj.Init*(*aPos*, *ExistentialRegistration*, *aCons*, *aTyp*);
 end;
destructor *EClusterObj.Done*;
 begin if *nClusterType* ≠ **nil then** *dispose*(*nClusterType*, *Done*);
 inherited Done;
 end;

991. 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` 852 ⟩ +≡
 CClusterPtr = ↑*CClusterObj*;
 CClusterObj = **object** (*ClusterObj*)
  nAntecedent: *PList*;
  constructor *Init*(**const** *aPos*: *Position*; *aAntec*, *aCons*: *PList*; *aTyp*: *TypePtr*);
  destructor *Done*; *virtual*;
 end ;

992. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *CClusterObj.Init*(**const** *aPos*: *Position*; *aAntec*, *aCons*: *PList*; *aTyp*: *TypePtr*);
 begin *ClusterObj.Init*(*aPos*, *ConditionalRegistration*, *aCons*, *aTyp*); *nAntecedent* ← *aAntec*;
 end;
destructor *CClusterObj.Done*;
 begin *dispose*(*nAntecedent*, *Done*); *inherited Done*;
 end;

993. 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.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  FClusterPtr = ↑*FClusterObj*;
  FClusterObj = **object** (*ClusterObj*)
    nClusterTerm: *TermPtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aTrm*: *TermPtr*; *aCons*: *PList*; *aTyp*: *TypePtr*);
    destructor *Done*; *virtual*;
  end ;

994. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *FClusterObj*.*Init*(**const** *aPos*: *Position*;
    aTrm: *TermPtr*; *aCons*: *PList*; *aTyp*: *TypePtr*);
  begin *ClusterObj*.*Init*(*aPos*, *FunctorialRegistration*, *aCons*, *aTyp*); *nClusterTerm* ← *aTrm*;
  end;
destructor *FClusterObj*.*Done*;
  begin if *nClusterTerm* ≠ **nil then** *Dispose*(*nClusterTerm*, *Done*);
  if *nClusterType* ≠ **nil then** *dispose*(*nClusterType*, *Done*);
  inherited Done;
  end;

995. Loci equality. This is used in identification registrations.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  LociEqualityPtr = ↑*LociEqualityObj*;
  LociEqualityObj = **object** (*mObject*)
    nEqPos: *Position*;
    nLeftLocus, *nRightLocus*: *LocusPtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aLeftLocus*, *aRightLocus*: *LocusPtr*);
    destructor *Done*; *virtual*;
  end ;

996. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *LociEqualityObj*.*Init*(**const** *aPos*: *Position*; *aLeftLocus*, *aRightLocus*: *LocusPtr*);
  begin *nEqPos* ← *aPos*; *nLeftLocus* ← *aLeftLocus*; *nRightLocus* ← *aRightLocus*;
  end;
destructor *LociEqualityObj*.*Done*;
  begin *Dispose*(*nLeftLocus*, *Done*); *dispose*(*nRightLocus*, *Done*);
  end;

997. 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.

⟨Publicly declared types in `wsmarticle.pas` 852⟩ +≡
  IdentifyRegistrationPtr = ↑*IdentifyRegistrationObj*;
  IdentifyRegistrationObj = **object** (*mObject*)
    nIdentifyPos: *Position*;
    nOriginPattern, *nNewPattern*: *PatternPtr*;
    nEqLociList: *PList*;
    constructor *Init*(**const** *aPos*: *Position*; *aNewPatt*, *aOrigPatt*: *PatternPtr*; *aEqList*: *PList*);
    destructor *Done*; *virtual*;
  end ;

998. Constructor.

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
constructor *IdentifyRegistrationObj.Init*(**const** *aPos*: *Position*;
    aNewPatt, *aOrigPatt*: *PatternPtr*; *aEqList*: *PList*);
  begin *nIdentifyPos* ← *aPos*; *nOriginPattern* ← *aOrigPatt*; *nNewPattern* ← *aNewPatt*;
  nEqLociList ← *aEqList*;
  end;
destructor *IdentifyRegistrationObj.Done*;
  begin *dispose*(*nOriginPattern*, *Done*); *dispose*(*nNewPattern*, *Done*);
  if *nEqLociList* ≠ **nil then** *dispose*(*nEqLociList*, *Done*);
  end;

999. Property registration. These were introduced in Mizar to facilitated registering "`sethood`" for types. Thus far, only the "`sethood`" property is handled in this registration.

⟨Publicly declared types in `wsmarticle.pas` 852⟩ +≡
  PropertyRegistrationPtr = ↑*PropertyRegistrationObj*;
  PropertyRegistrationObj = **object** (*mObject*)
    nPropertyPos: *Position*;
    nPropertySort: *PropertyKind*;
    constructor *Init*(**const** *aPos*: *Position*; *aKind*: *PropertyKind*);
    destructor *Done*; *virtual*;
  end ;

1000. Constructor.

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
constructor *PropertyRegistrationObj.Init*(**const** *aPos*: *Position*; *aKind*: *PropertyKind*);
  begin *nPropertyPos* ← *aPos*; *nPropertySort* ← *aKind*;
  end;
destructor *PropertyRegistrationObj.Done*;
  begin end;

1001. 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.

⟨ Publicly declared types in wsmarticle.pas 852 ⟩ +≡
  SethoodRegistrationPtr = ↑*SethoodRegistrationObj*;
  SethoodRegistrationObj = **object** (*PropertyRegistrationObj*)
    nSethoodType: *TypePtr*;
    nJustification: *JustificationPtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aKind*: *PropertyKind*; *aType*: *TypePtr*);
    destructor *Done*; *virtual*;
  end ;

1002. Constructor.

⟨ Implementation for wsmarticle.pas 854 ⟩ +≡
constructor *SethoodRegistrationObj.Init*(**const** *aPos*: *Position*;
    aKind: *PropertyKind*; *aType*: *TypePtr*);
  begin *inherited Init*(*aPos*, *aKind*); *nSethoodType* ← *aType*; *nJustification* ← **nil**;
  end;
destructor *SethoodRegistrationObj.Done*;
  begin *dispose*(*nSethoodType*, *Done*); *dispose*(*nJustification*, *Done*); *inherited Done*;
  end;

1003. 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.

⟨ Publicly declared types in wsmarticle.pas 852 ⟩ +≡
  ReduceRegistrationPtr = ↑*ReduceRegistrationObj*;
  ReduceRegistrationObj = **object** (*MObject*)
    nReducePos: *Position*;
    nOriginTerm, *nNewTerm*: *TermPtr*;
    constructor *Init*(**const** *aPos*: *Position*; *aOrigTerm*, *aNewTerm*: *TermPtr*);
    destructor *Done*; *virtual*;
  end ;

1004. Constructor.

⟨ Implementation for wsmarticle.pas 854 ⟩ +≡
constructor *ReduceRegistrationObj.Init*(**const** *aPos*: *Position*; *aOrigTerm*, *aNewTerm*: *TermPtr*);
  begin *nReducePos* ← *aPos*; *nOriginTerm* ← *aOrigTerm*; *nNewTerm* ← *aNewTerm*;
  end;
destructor *ReduceRegistrationObj.Done*;
  begin *dispose*(*nOriginTerm*, *Done*); *dispose*(*nNewTerm*, *Done*);
  end;

Section 21.10. HELPER FUNCTIONS

1005. Capitlization checks if the first character c is lowercase. If so, then set the leading character to be $c \leftarrow c - (ord(\texttt{\'a\'}) - ord(\texttt{\'A\'}))$. But it leaves the rest of the string untouched.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *CapitalizeName*(*aName* : *string*): *string*;
  begin *result* ← *aName*;
  if *aName*[1] ∈ [´a´ .. ´z´] **then** *dec*(*Result*[1], *ord*(´a´) − *ord*(´A´))
  end;

1006. Uncapitalizing works in the opposite direction, setting the first letter c of a string to be $c \leftarrow c + (ord(\texttt{\'a\'}) - ord(\texttt{\'A\'}))$. Observe capitalizing and uncapitalizing are "nearly inverses" of each other: $\mathit{CapitalizeName}(\mathit{UncapitalizeName}(\mathit{CapitalizeName}(s))) = \mathit{CapitalizeName}(s)$, and similarly we find $\mathit{UncapitalizeName}(\mathit{CapitalizeName}(\mathit{UncapitalizeName}(s))) = \mathit{UncapitalizeName}(s)$.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *UncapitalizeName*(*aName* : *string*): *string*;
  begin *result* ← *aName*;
  if *aName*[1] ∈ [´A´ .. ´Z´] **then** *inc*(*Result*[1], *ord*(´a´) − *ord*(´A´))
  end;

1007. We will be populating global variables tracking names of identifiers, modes, and other syntactic classes.

⟨ Global variables publicly declared in `wsmarticle.pas` 1007 ⟩ ≡
var *IdentifierName*, *AttributeName*, *StructureName*, *ModeName*, *PredicateName*, *FunctorName*,
    SelectorName, *LeftBracketName*, *RightBracketName*, *MMLIdentifierName*: **array of** *string*;

This code is used in section 850.

1008. We will want to initialize these global variables based on previous passes of the scanner.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *InitScannerNames*;
  var *i*, *lCnt*, *lNr*: *integer*; *lDct*: *text*; *lInFile*: *XMLInStreamPtr*; *lKind*, *lDummy*: *AnsiChar*;
    lString: *string*;
  begin ⟨ Populate global variables with XML entities 1009 ⟩;
  ⟨ Reset reserved keywords 1011 ⟩;
    { Identifiers }
  ⟨ Initialize identifier names from `.idx` file 1012 ⟩;
  end;

1009. We need to initialize the length for each of these arrays. Even a crude approximation works, like the total number of lines in the `.dct` file. Then we transform each line of the *lDct* (dictionary file) into appropriate entries of the relevant array.

⟨ Populate global variables with XML entities 1009 ⟩ ≡
  assign(*lDct*, *MizFileName* + ´`.dct`´); *reset*(*lDct*); *lCnt* ← 0;
  while ¬*seekEof*(*lDct*) **do**
    begin *readln*(*lDct*); *inc*(*lCnt*);
    end;
  setlength(*AttributeName*, *lCnt*); *setlength*(*StructureName*, *lCnt*); *setlength*(*ModeName*, *lCnt*);
  setlength(*PredicateName*, *lCnt*); *setlength*(*FunctorName*, *lCnt*); *setlength*(*SelectorName*, *lCnt*);
  setlength(*LeftBracketName*, *lCnt*); *setlength*(*RightBracketName*, *lCnt*);
  setlength(*MMLIdentifierName*, *lCnt*); *reset*(*lDct*);
  while ¬*seekEof*(*lDct*) **do**
    begin *readln*(*lDct*, *lKind*, *lNr*, *lDummy*, *lString*); ⟨ Store XML version of vocabulary word 1010 ⟩;
    end;
  close(*lDct*)

This code is used in section 1008.

1010. We have read in from the "`.dct`" file one line. The first 148 lines of a "`.dct`" file consists of the reserved keywords for Mizar. A random example of the last few lines of such a file look like:

```
A36 VECTSP_4
A37 ORDINAL1
A38 CARD_FIL
A39 RANKNULL
A40 VECTSP_1
A41 VECTSP_6
A42 VECTSP13
A43 ALGSTR_0
A44 HALLMAR1
A45 MATROID0
```

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*.

⟨ Store XML version of vocabulary word 1010 ⟩ ≡
  case *lKind* **of**
  ´`A`´: *MMLIdentifierName*[*lNr*] ← *QuoteStrForXML*(*lString*);
  ´`G`´: *StructureName*[*lNr*] ← *QuoteStrForXML*(*lString*);
  ´`M`´: *ModeName*[*lNr*] ← *QuoteStrForXML*(*lString*);
  ´`K`´: *LeftBracketName*[*lNr*] ← *QuoteStrForXML*(*lString*);
  ´`L`´: *RightBracketName*[*lNr*] ← *QuoteStrForXML*(*lString*);
  ´`O`´: *FunctorName*[*lNr*] ← *QuoteStrForXML*(*lString*);
  ´`R`´: *PredicateName*[*lNr*] ← *QuoteStrForXML*(*lString*);
  ´`U`´: *SelectorName*[*lNr*] ← *QuoteStrForXML*(*lString*);
  ´`V`´: *AttributeName*[*lNr*] ← *QuoteStrForXML*(*lString*);
  endcases

This code is used in section 1009.

1011. Preserve reserved keywords. We want to prevent the user from "overwriting" or "shadowing" the builtin primitive reserved words. This should probably be documented in the user-manual somewhere. The reserved words are: "`strict`", "`set`","`=`", and the brackets `[]`, braces `{}`, and parentheses `()`. Curiously, "`object`" is not considered a 'primitive' worth preserving.

⟨ Reset reserved keywords 1011 ⟩ ≡

```
AttributeName[StrictSym] ← 'strict'; ModeName[SetSym] ← 'set';
PredicateName[EqualitySym] ← '='; LeftBracketName[SquareBracket] ← '[';
LeftBracketName[CurlyBracket] ← '{'; LeftBracketName[RoundedBracket] ← '(';
RightBracketName[SquareBracket] ← ']'; RightBracketName[CurlyBracket] ← '}';
RightBracketName[RoundedBracket] ← ')'
```

This code is used in section 1008.

1012. The `.idx` file provides numbers for the local labels and article names referenced in an article.

⟨ Initialize identifier names from `.idx` file 1012 ⟩ ≡

```
assign(lDct, MizFileName + '.idx'); reset(lDct); lCnt ← 0;
while ¬seekEof(lDct) do
  begin readln(lDct); inc(lCnt);
  end;
close(lDct);
setlength(IdentifierName, lCnt); IdentifierName[0] ← '';
lInFile ← new(XMLInStreamPtr, OpenFile(MizFileName + '.idx')); lInFile↑.NextElementState;
lInFile↑.NextElementState;
while (lInFile.nState = eStart) ∧ (lInFile.nElName = XMLElemName[elSymbol]) do
  begin lNr ← lInFile↑.GetIntAttr('nr'); lString ← lInFile↑.GetAttr('name');
  IdentifierName[lNr] ← lString; lInFile↑.NextElementState; lInFile↑.NextElementState;
  end;
dispose(lInFile, Done)
```

This code is used in section 1008.

1013. We will want to obtain the name for an article ID number, provided it is a legal number (i.e., less than the dictionary for article ID numbers). This function looks up its entry in the *IdentifierName* array.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
function IdentRepr(aIdNr : integer): string;
  begin mizassert(2000, aIdNr ≤ length(IdentifierName));
  if aIdNr > 0 then IdentRepr ← IdentifierName[aIdNr]
  else IdentRepr ← '';
  end;
```

Section 21.11. WRITING WSM XML FILES

1014.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
 OutWSMizFilePtr = ↑*OutWSMizFileObj*;
 OutWSMizFileObj = **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*(*aSegm* : *ImplicitlyQualifiedSegmentPtr*); *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 ;

1015. Constructor. The constructor *OutWSMizFileObj.OpenFileWithXSL* is not used anywhere, nor is the associated "`wsmiz.xml`" file present anywhere.

Importantly, the *nMizarAppearance* field controls whether the XML generated includes the raw lexeme string as an attribute in the XML elements or not.

The constructor *OpenFileWithXSL* is never used. The XML stylesheet `wsmiz.xml` does not seem to be present in the Mizar distribution.

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
constructor *OutWSMizFileObj.OpenFile*(**const** *aFileName*: *string*);
  begin *inherited OpenFile*(*aFileName*); *nMizarAppearance* ← *false*;
  nDisplayInformationOnScreen ← *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 ← *false*;
  end;
destructor *OutWSMizFileObj.Done*;
  begin *inherited Done*;
  end;

1016. We can write the XML for a *wsTextProper* object (§856). This writes out the start tag, the children, and the end-tag for the "text proper" and its contents. The RNG compact schema for this looks like:

```
TextProper = element Text-Proper {
  attribute idnr { xsd:integer },
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Item*
}
```

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
procedure *OutWSMizFileObj.Out_TextProper*(*aWSTextProper* : *WSTextProperPtr*);
  var *i*: *integer*;
  begin with *aWSTextProper*↑ **do**
    begin { Write the start-tag }
    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*↑.*Count* − 1 **do** *Out_Item*(*nItems*.*Items*↑[*i*]); { ...then write the children }
    Out_XElEnd(*BlockName*[*blMain*]);
    end;
  end;

1017. Writing a block out as XML works similarly: write the start-tag, then its children elements, then the end-tag.

```
Block = element Block {
  attribute kind { "Text-Proper" | "Now-Reasoning"
    | "Hereby-Reasoning" | "Definitional-Block"
    | "Notation-Block" | "Registration-Block" | "Case"
    | "Suppose" | "Scheme-Block" },
  attribute idnr { xsd:integer },
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Item*
}
```

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *OutWSMizFileObj.Out_Block*(*aWSBlock* : *WSBlockPtr*);
  var *i*: *integer*;
  begin with *aWSBlock*↑ **do**
    begin { write the start-tag }
    Out_XElStart(*XMLElemName*[*elBlock*]);
    Out_XAttr(*XMLAttrName*[*atKind*], *BlockName*[*nBlockKind*]); *CurPos* ← *nBlockPos*;
    Out_PosAsAttrs(*nBlockPos*); *Out_XIntAttr*(*XMLAttrName*[*atPosLine*], *nBlockEndPos.Line*);
    Out_XIntAttr(*XMLAttrName*[*atPosCol*], *nBlockEndPos.Col*); *Out_XAttrEnd*;
    for $i \leftarrow 0$ **to** *nItems*↑.*Count* − 1 **do**
      begin *Out_Item*(*nItems*↑.*Items*↑[*i*]); **end**; { Then write the children }
    Out_XElEnd(*XMLElemName*[*elBlock*]);
    end;
  end;

1018. Writing a term list to XML amounts to just writing the terms as XML elements. They will be contained in a parent element, so there will be no ambiguity in their role.

```
Term-List = ( Term* )
```

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *OutWSMizFileObj.Out_TermList*(*aTrmList* : *PList*);
  var *i*: *integer*;
  begin for $i \leftarrow 0$ **to** *aTrmList*↑.*Count* − 1 **do** *Out_Term*(*aTrmList*↑.*Items*↑[*i*]);
  end;

1019. The XML for an adjective boils down to two cases:

Case 1 (negated attribute). Write a `<NegatedAdjective>` tag around the XML produced from case 2 for the positive version of the attribute.

Case 2 (positive attribute). Write the adjective, and its children are the [term] arguments to the adjective (if any — if there are none, then an empty-element will be produced).

```
PositiveAdjective = element Adjective {
  attribute nr { xsd:integer },
  attribute name { text },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer }
  Term*
}
Adjective = PositiveAdjective | element NegatedAdjective {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  PositiveAdjective
}
```

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
procedure OutWSMizFileObj.Out_Adjective(aAttr : AdjectiveExpressionPtr);
  begin case aAttr↑.nAdjectiveSort of
  wsAdjective: begin Out_XElStart(XMLElemName[elAdjective]);
    with AdjectivePtr(aAttr)↑ do
      begin Out_XIntAttr(XMLAttrName[atNr], nAdjectiveSymbol);
      if nMizarAppearance then
        Out_XAttr(XMLAttrName[atSpelling], AttributeName[nAdjectiveSymbol]);
      Out_PosAsAttrs(nAdjectivePos);
      if nArgs↑.Count = 0 then Out_XElEnd0
      else begin Out_XAttrEnd; Out_TermList(nArgs); Out_XElEnd(XMLElemName[elAdjective]);
        end;
      end;
    end;
  wsNegatedAdjective: begin Out_XElStart(XMLElemName[elNegatedAdjective]);
    with NegatedAdjectivePtr(aAttr)↑ do
      begin Out_PosAsAttrs(nAdjectivePos); Out_XAttrEnd; Out_Adjective(nArg);
      end;
    Out_XElEnd(XMLElemName[elNegatedAdjective]);
    end;
  endcases;
  end;
```

1020. Writing an adjective list to XML amounts to stuffing all the adjectives into an element. If there are no adjectives, it is the empty-element.

```
Adjective-Cluster = element Adjective-Cluster {
  attribute count { xsd:integer },
  Adjective*
}
```

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

procedure *OutWSMizFileObj.Out_AdjectiveList*(*aCluster* : *PList*);
  var *i*: *integer*;
  begin *Out_XElStart*(*XMLElemName*[*elAdjectiveCluster*]);
  if *aCluster*↑.*Count* = 0 **then**
    begin *Out_XElEnd0*; *exit*;
    end;
  Out_XAttrEnd;
  with *aCluster*↑ **do**
    for $i \leftarrow 0$ **to** $Count - 1$ **do** *Out_Adjective*(*Items*↑[*i*]);
  Out_XElEnd(*XMLElemName*[*elAdjectiveCluster*]);
  end;

Subsection 21.11.1. Emitting XML for types

1021. Writing the XML for a Mizar type.

```
StandardType = element Standard-Type {
  attribute nr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term*
}
StructureType = element Structure-Type {
  attribute nr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term*
}
ClusteredType = element Clustered-Type {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Adjective-Cluster,
  Type,
}
Type = StandardType | StructureType | ClusteredType
```

define *print_arguments*(#) ≡
  if *nArgs*↑.*Count* = 0 **then** *Out_XElEnd0*
  else begin *Out_XAttrEnd*; *Out_TermList*(*nArgs*); *Out_XElEnd*(*TypeName*[#]);
   end

⟨ Implementation for wsmarticle.pas 854 ⟩ +≡
procedure *OutWSMizFileObj*.*Out_Type*(*aTyp* : *TypePtr*);
 begin with *aTyp*↑ **do**
  case *aTyp*↑.*nTypeSort* **of**
  wsStandardType: **with** *StandardTypePtr*(*aTyp*)↑ **do**
    begin *Out_XElStart*(*TypeName*[*wsStandardType*]);
    Out_XIntAttr(*XMLAttrName*[*atNr*], *nModeSymbol*);
    if *nMizarAppearance* **then** *Out_XAttr*(*XMLAttrName*[*atSpelling*], *ModeName*[*nModeSymbol*]);
    Out_PosAsAttrs(*nTypePos*); *print_arguments*(*wsStandardType*);
    end;
  wsStructureType: **with** *StructTypePtr*(*aTyp*)↑ **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*)↑ **do**
    begin *Out_XElStart*(*TypeName*[*wsClusteredType*]); *Out_PosAsAttrs*(*nTypePos*); *Out_XAttrEnd*;
    Out_AdjectiveList(*nAdjectiveCluster*); *Out_Type*(*nType*);
    Out_XElEnd(*TypeName*[*wsClusteredType*]);
    end;
  wsErrorType: **begin** *Out_XElWithPos*(*TypeName*[*wsErrorType*], *nTypePos*);
   end;
  endcases;

end;

1022. Printing a variable as an XML element.

```
Variable = element Variable {
  attribute idnr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer }
}
```

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *OutWSMizFileObj.Out_Variable*(*aVar* : *VariablePtr*);
  begin with *aVar*↑ **do**
    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;

1023. Variables introduced using "`reserve`" are just printed out like any other variable.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *OutWSMizFileObj.Out_ReservedVariable*(*aVar* : *VariablePtr*);
  begin *Out_Variable*(*aVar*);
  end;

1024. Implicitly qualified variables (i.e., variables which are `reserved` with a type, then used in, e.g., a quantified formula) are just variables appearing as children of an "implicitly qualified" XML element.

```
  VariableSegment |= element Implicitly-Qualified-Segment {
    attribute col { xsd:integer },
    attribute line { xsd:integer },
    Variable
  }
```

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *OutWSMizFileObj.Out_ImplicitlyQualifiedVariable*(*aSegm* : *ImplicitlyQualifiedSegmentPtr*);
  begin *Out_XElStart*(*SegmentKindName*[*ikImplQualifiedSegm*]); *Out_PosAsAttrs*(*aSegm*↑.*nSegmPos*);
  Out_XAttrEnd; *Out_Variable*(*aSegm*↑.*nIdentifier*);
  Out_XElEnd(*SegmentKindName*[*ikImplQualifiedSegm*]);
  end;

1025. Qualified variable segments are either implicitly qualified (hence we use the previous function) or explicitly qualified (which look like "⟨*variable list*⟩ `being` ⟨*type*⟩").

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
}
```

⟨Implementation for `wsmarticle.pas` 854⟩ +≡

procedure *OutWSMizFileObj.Out_VariableSegment*(*aSegm* : *QualifiedSegmentPtr*);
  var *i*: *integer*;
  begin case *aSegm*↑.*nSegmentSort* **of**
  ikImplQualifiedSegm: *Out_ImplicitlyQualifiedVariable*(*ImplicitlyQualifiedSegmentPtr*(*aSegm*));
  ikExplQualifiedSegm: **with** *ExplicitlyQualifiedSegmentPtr*(*aSegm*)↑ **do**
      begin *Out_XElStart*(*SegmentKindName*[*ikExplQualifiedSegm*]); *Out_PosAsAttrs*(*nSegmPos*);
      Out_XAttrEnd; *Out_XElStart0*(*XMLElemName*[*elVariables*]);
      for $i \leftarrow 0$ **to** *nIdentifiers*↑.*Count* − 1 **do** *Out_Variable*(*nIdentifiers*↑.*Items*↑[*i*]);
      Out_XElEnd(*XMLElemName*[*elVariables*]); *Out_Type*(*nType*);
      Out_XElEnd(*SegmentKindName*[*ikExplQualifiedSegm*]);
      end;
  endcases;
  end;

1026. Private predicates have the XML schema

```
Private-Predicate-Formula = element Private-Predicate-Formula {
  attribute idnr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term-List?
}
```

⟨Implementation for `wsmarticle.pas` 854⟩ +≡

procedure *OutWSMizFileObj.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*↑.*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

1027. The XML schema for formulas looks something like:

```
Formula = NegatedFormula
| ConjunctiveFormula
| DisjunctiveFormula
| ConditionalFormula
| BiconditionalFormula
| FlexaryConjunctiveFormula
| FlexaryDisjunctiveFormula
| Predicative-Formula
| RightSideOf-Predicative-Formula
| Multi-Predicative-Formula
| Attributive-Formula
| Qualifying-Formula
| Universal-Quantifier-Formula
| Existential-Quantifier-Formula
| element Contradiction {
    attribute col { xsd:integer },
    attribute line { xsd:integer } }
| element Thesis {
    attribute col { xsd:integer },
    attribute line { xsd:integer } }
| element Formula-Error {
    attribute col { xsd:integer },
    attribute line { xsd:integer } }
```

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
procedure *OutWSMizFileObj.Out_Formula*(*aFrm* : *FormulaPtr*);
  var *i*: *integer*;
  begin case *aFrm*↑.*nFormulaSort* **of**
  wsNegatedFormula: ⟨Emit XML for negated formula (WSM) 1028⟩;
  wsConjunctiveFormula: ⟨Emit XML for conjunction (WSM) 1029⟩;
  wsDisjunctiveFormula: ⟨Emit XML for disjunction (WSM) 1030⟩;
  wsConditionalFormula: ⟨Emit XML for conditional formula (WSM) 1031⟩;
  wsBiconditionalFormula: ⟨Emit XML for biconditional formula (WSM) 1032⟩;
  wsFlexaryConjunctiveFormula: ⟨Emit XML for flexary-conjunction (WSM) 1033⟩;
  wsFlexaryDisjunctiveFormula: ⟨Emit XML for flexary-disjunction (WSM) 1034⟩;
  wsPredicativeFormula: ⟨Emit XML for predicative formula (WSM) 1035⟩;
  wsRightSideOfPredicativeFormula: ⟨Emit XML for right-side of predicative formula (WSM) 1036⟩;
  wsMultiPredicativeFormula: ⟨Emit XML for multi-predicative formula (WSM) 1037⟩;
  wsPrivatePredicateFormula: *Out_PrivatePredicativeFormula*(*PrivatePredicativeFormulaPtr*(*aFrm*));
  wsAttributiveFormula: ⟨Emit XML for attributive formula (WSM) 1038⟩;
  wsQualifyingFormula: ⟨Emit XML for qualifying formula (WSM) 1039⟩;
  wsUniversalFormula: ⟨Emit XML for universal formula (WSM) 1040⟩;
  wsExistentialFormula: ⟨Emit XML for existential formula (WSM) 1041⟩;
  wsContradiction: **begin** *Out_XElWithPos*(*FormulaName*[*wsContradiction*], *aFrm*↑.*nFormulaPos*);
    end;
  wsThesis: **begin** *Out_XElWithPos*(*FormulaName*[*wsThesis*], *aFrm*↑.*nFormulaPos*);
    end;
  wsErrorFormula: **begin** *Out_XElWithPos*(*FormulaName*[*wsErrorFormula*], *aFrm*↑.*nFormulaPos*);
    end;
  endcases;
  end;

1028.

```
NegatedFormula = element Negated-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula
}
```

⟨ Emit XML for negated formula (WSM) 1028 ⟩ ≡
begin *Out_XElStart*(*FormulaName*[*wsNegatedFormula*]); *Out_PosAsAttrs*(*aFrm*↑.*nFormulaPos*);
Out_XAttrEnd; *Out_Formula*(*NegativeFormulaPtr*(*aFrm*)↑.*nArg*);
Out_XElEnd(*FormulaName*[*wsNegatedFormula*]);
end

This code is used in section 1027.

1029.

```
ConjunctiveFormula = element Conjunctive-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula,
  Formula
}
```

⟨ Emit XML for conjunction (WSM) 1029 ⟩ ≡
begin *Out_XElStart*(*FormulaName*[*wsConjunctiveFormula*]); *Out_PosAsAttrs*(*aFrm*↑.*nFormulaPos*);
Out_XAttrEnd; *Out_Formula*(*BinaryFormulaPtr*(*aFrm*)↑.*nLeftArg*);
Out_Formula(*BinaryFormulaPtr*(*aFrm*)↑.*nRightArg*);
Out_XElEnd(*FormulaName*[*wsConjunctiveFormula*]);
end

This code is used in section 1027.

1030.

```
DisjunctiveFormula = element Disjunctive-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula,
  Formula
}
```

⟨ Emit XML for disjunction (WSM) 1030 ⟩ ≡
begin *Out_XElStart*(*FormulaName*[*wsDisjunctiveFormula*]); *Out_PosAsAttrs*(*aFrm*↑.*nFormulaPos*);
Out_XAttrEnd; *Out_Formula*(*BinaryFormulaPtr*(*aFrm*)↑.*nLeftArg*);
Out_Formula(*BinaryFormulaPtr*(*aFrm*)↑.*nRightArg*);
Out_XElEnd(*FormulaName*[*wsDisjunctiveFormula*]);
end

This code is used in section 1027.

1031.

```
ConditionalFormula = element Conditional-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula,
  Formula
}
```

⟨ Emit XML for conditional formula (WSM) 1031 ⟩ ≡
begin *Out_XElStart*(*FormulaName*[*wsConditionalFormula*]); *Out_PosAsAttrs*(*aFrm*↑.*nFormulaPos*);
Out_XAttrEnd; *Out_Formula*(*BinaryFormulaPtr*(*aFrm*)↑.*nLeftArg*);
Out_Formula(*BinaryFormulaPtr*(*aFrm*)↑.*nRightArg*);
Out_XElEnd(*FormulaName*[*wsConditionalFormula*]);
end

This code is used in section 1027.

1032.

```
BiconditionalFormula = element Biconditional-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula,
  Formula
}
```

⟨ Emit XML for biconditional formula (WSM) 1032 ⟩ ≡
begin *Out_XElStart*(*FormulaName*[*wsBiconditionalFormula*]); *Out_PosAsAttrs*(*aFrm*↑.*nFormulaPos*);
Out_XAttrEnd; *Out_Formula*(*BinaryFormulaPtr*(*aFrm*)↑.*nLeftArg*);
Out_Formula(*BinaryFormulaPtr*(*aFrm*)↑.*nRightArg*);
Out_XElEnd(*FormulaName*[*wsBiconditionalFormula*]);
end

This code is used in section 1027.

1033.

```
FlexaryConjunctiveFormula = element FlexaryConjunctive-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula,
  Formula
}
```

⟨ Emit XML for flexary-conjunction (WSM) 1033 ⟩ ≡
begin *Out_XElStart*(*FormulaName*[*wsFlexaryConjunctiveFormula*]);
Out_PosAsAttrs(*aFrm*↑.*nFormulaPos*); *Out_XAttrEnd*;
Out_Formula(*BinaryFormulaPtr*(*aFrm*)↑.*nLeftArg*);
Out_Formula(*BinaryFormulaPtr*(*aFrm*)↑.*nRightArg*);
Out_XElEnd(*FormulaName*[*wsFlexaryConjunctiveFormula*]);
end

This code is used in section 1027.

1034.

```
FlexaryDisjunctiveFormula = element FlexaryDisjunctive-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula,
  Formula
}
```

⟨ Emit XML for flexary-disjunction (WSM) 1034 ⟩ ≡
begin *Out_XElStart*(*FormulaName*[*wsFlexaryDisjunctiveFormula*]);
Out_PosAsAttrs(*aFrm*↑.*nFormulaPos*); *Out_XAttrEnd*;
Out_Formula(*BinaryFormulaPtr*(*aFrm*)↑.*nLeftArg*);
Out_Formula(*BinaryFormulaPtr*(*aFrm*)↑.*nRightArg*);
Out_XElEnd(*FormulaName*[*wsFlexaryDisjunctiveFormula*]);
end

This code is used in section 1027.

1035.

```
Predicative-Formula = element Predicative-Formula {
  attribute nr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  element Arguments { Term-List? },
  element Arguments { Term-List? }
}
```

⟨ Emit XML for predicative formula (WSM) 1035 ⟩ ≡
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*↑.*Count* $=0$ **then** *Out_XEl1*(*XMLElemName*[*elArguments*])
else begin *Out_XElStart0*(*XMLElemName*[*elArguments*]); *Out_TermList*(*nLeftArgs*);
Out_XElEnd(*XMLElemName*[*elArguments*]);
end;
if *nRightArgs*↑.*Count* $=0$ **then** *Out_XEl1*(*XMLElemName*[*elArguments*])
else begin *Out_XElStart0*(*XMLElemName*[*elArguments*]); *Out_TermList*(*nRightArgs*);
Out_XElEnd(*XMLElemName*[*elArguments*]);
end;
Out_XElEnd(*FormulaName*[*wsPredicativeFormula*]);
end

This code is used in section 1027.

1036.

```
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? }
}
```

⟨Emit XML for right-side of predicative formula (WSM) 1036⟩ ≡
  with *RightSideOfPredicativeFormulaPtr*(*aFrm*)↑ **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*↑.*Count* = 0 **then** *Out_XEl1*(*XMLElemName*[*elArguments*])
    else begin *Out_XElStart0*(*XMLElemName*[*elArguments*]); *Out_TermList*(*nRightArgs*);
      Out_XElEnd(*XMLElemName*[*elArguments*]);
      end;
    Out_XElEnd(*FormulaName*[*wsRightSideOfPredicativeFormula*])
    end

This code is used in section 1027.

1037.

```
Multi-Predicative-Formula = element Multi-Predicative-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Formula*
}
```

⟨Emit XML for multi-predicative formula (WSM) 1037⟩ ≡
  with *MultiPredicativeFormulaPtr*(*aFrm*)↑ **do**
    begin *Out_XElStart*(*FormulaName*[*wsMultiPredicativeFormula*]);
    Out_PosAsAttrs(*aFrm*↑.*nFormulaPos*); *Out_XAttrEnd*;
    for $i \leftarrow 0$ **to** *nScraps*.*Count* − 1 **do** *Out_Formula*(*nScraps*↑.*Items*↑[*i*]);
    Out_XElEnd(*FormulaName*[*wsMultiPredicativeFormula*])
    end

This code is used in section 1027.

1038.

```
Attributive-Formula = element Attributive-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term,
  Adjective-Cluster.element
}
```

⟨ Emit XML for attributive formula (WSM) 1038 ⟩ ≡
  with *AttributiveFormulaPtr*(*aFrm*)↑ **do**
    begin *Out_XElStart*(*FormulaName*[*wsAttributiveFormula*]); *Out_PosAsAttrs*(*nFormulaPos*);
    Out_XAttrEnd; *Out_Term*(*nSubject*); *Out_AdjectiveList*(*nAdjectives*);
    Out_XElEnd(*FormulaName*[*wsAttributiveFormula*]);
    end

This code is used in section 1027.

1039.

```
Qualifying-Formula = element Qualifying-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term,
  Type,
  Formula
}
```

⟨ Emit XML for qualifying formula (WSM) 1039 ⟩ ≡
  with *QualifyingFormulaPtr*(*aFrm*)↑ **do**
    begin *Out_XElStart*(*FormulaName*[*wsQualifyingFormula*]); *Out_PosAsAttrs*(*nFormulaPos*);
    Out_XAttrEnd; *Out_Term*(*nSubject*); *Out_Type*(*nType*);
    Out_XElEnd(*FormulaName*[*wsQualifyingFormula*]);
    end

This code is used in section 1027.

1040.

```
Universal-Quantifier-Formula = element Universal-Quantifier-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Variable-Segment,
  Formula
}
```

⟨ Emit XML for universal formula (WSM) 1040 ⟩ ≡
  with *QuantifiedFormulaPtr*(*aFrm*)↑ **do**
    begin *Out_XElStart*(*FormulaName*[*wsUniversalFormula*]); *Out_PosAsAttrs*(*nFormulaPos*);
    Out_XAttrEnd; *Out_VariableSegment*(*QuantifiedFormulaPtr*(*aFrm*)↑.*nSegment*);
    Out_Formula(*QuantifiedFormulaPtr*(*aFrm*)↑.*nScope*);
    Out_XElEnd(*FormulaName*[*wsUniversalFormula*]);
    end

This code is used in section 1027.

1041.

```
Existential-Quantifier-Formula = element Existential-Quantifier-Formula {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Variable-Segment,
  Formula
}
```

⟨ Emit XML for existential formula (WSM) 1041 ⟩ ≡
  with *QuantifiedFormulaPtr*(*aFrm*)↑ **do**
    begin *Out_XElStart*(*FormulaName*[*wsExistentialFormula*]); *Out_PosAsAttrs*(*nFormulaPos*);
    Out_XAttrEnd; *Out_VariableSegment*(*QuantifiedFormulaPtr*(*aFrm*)↑.*nSegment*);
    Out_Formula(*QuantifiedFormulaPtr*(*aFrm*)↑.*nScope*);
    Out_XElEnd(*FormulaName*[*wsExistentialFormula*]);
    end

This code is used in section 1027.

Subsection 21.11.3. Emitting XML for Terms

1042. We begin with simple terms.

```
Term |= element Simple-Term {
  attribute idnr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer }
}
```

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *OutWSMizFileObj.Out_SimpleTerm*(*aTrm* : *SimpleTermPtr*);
  begin *Out_XElStart*(*TermName*[*wsSimpleTerm*]);
  Out_XIntAttr(*XMLAttrName*[*atIdNr*], *aTrm*↑.*nIdent*);
  if *nMizarAppearance* **then** *Out_XAttr*(*XMLAttrName*[*atSpelling*], *IdentRepr*(*aTrm*↑.*nIdent*));
  Out_PosAsAttrs(*aTrm*↑.*nTermPos*); *Out_XElEnd0*;
  end;

1043. Terms: Private functors.

```
Term |= element Private-Functor-Term {
  attribute idnr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  element Arguments { Term-List }?
}
```

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
procedure *OutWSMizFileObj.Out_PrivateFunctorTerm*(*aTrm* : *PrivateFunctorTermPtr*);
  begin with *PrivateFunctorTermPtr*(*aTrm*)↑ **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*↑.*Count* = 0 **then** *Out_XElEnd0*
    else begin *Out_XAttrEnd*; *Out_TermList*(*nArgs*); *Out_XElEnd*(*TermName*[*wsPrivateFunctorTerm*]);
      end;
    end;
  end;

1044. Terms: internal selectors.

```
Term |= element Internal-Selector-Term {
  attribute nr { text },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer }
}
```

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
procedure *OutWSMizFileObj.Out_InternalSelectorTerm*(*aTrm* : *InternalSelectorTermPtr*);
  begin with *aTrm*↑ **do**
    begin *Out_XElStart*(*TermName*[*wsInternalSelectorTerm*]);
    Out_XIntAttr(*XMLAttrName*[*atNr*], *nSelectorSymbol*);
    if *nMizarAppearance* **then** *Out_XAttr*(*XMLAttrName*[*atSpelling*], *SelectorName*[*nSelectorSymbol*]);
    Out_PosAsAttrs(*nTermPos*); *Out_XElEnd0*;
    end;
  end;

1045. Terms: numerals, anaphoric "it", error.

```
Term |= element Numeral {
    attribute number { xsd:int },
    attribute col { xsd:integer },
    attribute line { xsd:integer }
  }
Term |= element It-Term {
    attribute col { xsd:integer },
    attribute line { xsd:integer }
  }
Term |= element Error-Term { }
```

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
procedure *OutWSMizFileObj.Out_Term*(*aTrm* : *TermPtr*);
  var *i*: *integer*;
  begin case *aTrm*↑.*nTermSort* **of**
  wsPlaceholderTerm: ⟨Emit XML for placeholder (WSM) 1046⟩;
  wsSimpleTerm: *Out_SimpleTerm*(*SimpleTermPtr*(*aTrm*));
  wsNumeralTerm: **begin** ; *Out_XElStart*(*TermName*[*wsNumeralTerm*]);
    Out_XIntAttr(*XMLAttrName*[*atNumber*], *NumeralTermPtr*(*aTrm*)↑.*nValue*);
    Out_PosAsAttrs(*aTrm*↑.*nTermPos*); *Out_XElEnd0*;
    end;
  wsInfixTerm: ⟨Emit XML for infix term (WSM) 1047⟩;
  wsCircumfixTerm: ⟨Emit XML for circumfix term (WSM) 1048⟩;
  wsPrivateFunctorTerm: *Out_PrivateFunctorTerm*(*PrivateFunctorTermPtr*(*aTrm*));
  wsAggregateTerm: ⟨Emit XML for aggregate term (WSM) 1049⟩;
  wsSelectorTerm: ⟨Emit XML for selector term (WSM) 1050⟩;
  wsInternalSelectorTerm: *Out_InternalSelectorTerm*(*InternalSelectorTermPtr*(*aTrm*));
  wsForgetfulFunctorTerm: ⟨Emit XML for forgetful functor (WSM) 1051⟩;
  wsInternalForgetfulFunctorTerm: ⟨Emit XML for internal forgetful functor (WSM) 1052⟩;
  wsFraenkelTerm: ⟨Emit XML for Fraenkel term (WSM) 1053⟩;
  wsSimpleFraenkelTerm: ⟨Emit XML for simple Fraenkel term (WSM) 1054⟩;
  wsQualificationTerm: ⟨Emit XML for qualification term (WSM) 1055⟩;
  wsExactlyTerm: ⟨Emit XML for exactly qualification term (WSM) 1056⟩;
  wsGlobalChoiceTerm: ⟨Emit XML for global choice term (WSM) 1057⟩;
  wsItTerm: *Out_XElWithPos*(*TermName*[*wsItTerm*], *aTrm*↑.*nTermPos*);
  wsErrorTerm: *Out_XEl1*(*TermName*[*wsErrorTerm*]);
  endcases;
  end;

1046. Terms: placeholders.

```
Term |= element Placeholder-Term {
  attribute nr { text },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer }
}
```

⟨ Emit XML for placeholder (WSM) 1046 ⟩ ≡

```
begin Out_XElStart(TermName[wsPlaceholderTerm]);
Out_XIntAttr(XMLAttrName[atNr], PlaceholderTermPtr(aTrm)↑.nLocusNr);
if nMizarAppearance then Out_XAttr(XMLAttrName[atSpelling],
      QuoteStrForXML(PlaceHolderName[PlaceholderTermPtr(aTrm)↑.nLocusNr]));
Out_PosAsAttrs(aTrm↑.nTermPos); Out_XElEnd0;
end
```

This code is used in section 1045.

1047. Terms: infixed.

```
Term |= element Infix-Term {
  attribute nr { text },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  element Arguments { Term-List? },
  element Arguments { Term-List? }
}
```

⟨ Emit XML for infix term (WSM) 1047 ⟩ ≡

```
with InfixTermPtr(aTrm)↑ 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↑.Count = 0 then Out_XEl1(XMLElemName[elArguments])
  else begin Out_XElStart0(XMLElemName[elArguments]); Out_TermList(nLeftArgs);
    Out_XElEnd(XMLElemName[elArguments]);
    end;
  if nRightArgs↑.Count = 0 then Out_XEl1(XMLElemName[elArguments])
  else begin Out_XElStart0(XMLElemName[elArguments]); Out_TermList(nRightArgs);
    Out_XElEnd(XMLElemName[elArguments]);
    end;
  Out_XElEnd(TermName[wsInfixTerm]);
  end
```

This code is used in section 1045.

1048. Terms: brackets.

```
Term |= element Circumfix-Term {
  attribute nr { text },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  element Right-Circumflex-Symbol {
    attribute nr { text },
    attribute spelling { text }?,
    attribute col { xsd:integer },
    attribute line { xsd:integer }
  },
  element Arguments { Term-List? }
}
```

⟨ Emit XML for circumfix term (WSM) 1048 ⟩ ≡
  with *CircumfixTermPtr*(*aTrm*)↑ **do**
    begin *Out_XElStart*(*TermName*[*wsCircumfixTerm*]);
    Out_XIntAttr(*XMLAttrName*[*atNr*], *nLeftBracketSymbol*);
    if *nMizarAppearance* **then**
      Out_XAttr(*XMLAttrName*[*atSpelling*], *LeftBracketName*[*nLeftBracketSymbol*]);
    Out_PosAsAttrs(*nTermPos*); *Out_XAttrEnd*;
    Out_XElStart(*XMLElemName*[*elRightCircumflexSymbol*]);
    Out_XIntAttr(*XMLAttrName*[*atNr*], *nRightBracketSymbol*);
    if *nMizarAppearance* **then**
      Out_XAttr(*XMLAttrName*[*atSpelling*], *RightBracketName*[*nRightBracketSymbol*]);
    Out_PosAsAttrs(*nTermPos*); *Out_XElEnd0*; *Out_TermList*(*nArgs*);
    Out_XElEnd(*TermName*[*wsCircumfixTerm*]);
    end

This code is used in section 1045.

1049. Terms: structure instances.

```
Term |= element Aggregate-Term {
  attribute nr { text },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  element Arguments { Term-List }?
}
```

⟨ Emit XML for aggregate term (WSM) 1049 ⟩ ≡
  with *AggregateTermPtr*(*aTrm*)↑ **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*↑.*Count* = 0 **then** *Out_XElEnd0*
    else begin *Out_XAttrEnd*; *Out_TermList*(*nArgs*); *Out_XElEnd*(*TermName*[*wsAggregateTerm*]);
      end;
    end

This code is used in section 1045.

1050. Terms: selectors.

```
Term |= element Selector-Term {
  attribute nr { text },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term
}
```

⟨ Emit XML for selector term (WSM) 1050 ⟩ ≡
with *SelectorTermPtr*(*aTrm*)↑ **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*]);
end

This code is used in section 1045.

1051. Terms: forgetful functors.

```
Term |= element Forgetful-Functor-Term {
  attribute nr { text },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term
}
```

⟨ Emit XML for forgetful functor (WSM) 1051 ⟩ ≡
with *ForgetfulFunctorTermPtr*(*aTrm*)↑ **do**
begin *Out_XElStart*(*TermName*[*wsForgetfulFunctorTerm*]);
Out_XIntAttr(*XMLAttrName*[*atNr*], *nStructSymbol*);
if *nMizarAppearance* **then** *Out_XAttr*(*XMLAttrName*[*atSpelling*], *StructureName*[*nStructSymbol*]);
Out_PosAsAttrs(*nTermPos*); *Out_XAttrEnd*; *Out_Term*(*nArg*);
Out_XElEnd(*TermName*[*wsForgetfulFunctorTerm*]);
end

This code is used in section 1045.

1052. Terms: internal forgetful functors.

```
Term |= element Internal-Forgetful-Functor-Term {
  attribute nr { text },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
}
```

⟨ Emit XML for internal forgetful functor (WSM) 1052 ⟩ ≡
with *InternalForgetfulFunctorTermPtr*(*aTrm*)↑ **do**
begin *Out_XElStart*(*TermName*[*wsInternalForgetfulFunctorTerm*]);
Out_XIntAttr(*XMLAttrName*[*atNr*], *nStructSymbol*); *Out_PosAsAttrs*(*nTermPos*); *Out_XElEnd0*;
end

This code is used in section 1045.

1053. Terms: Fraenkel operators.

```
Term |= element Fraenkel-Term {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Variable-Segment*,
  Term,
  Formula
}
```

⟨ Emit XML for Fraenkel term (WSM) 1053 ⟩ ≡
  with *FraenkelTermPtr*(*aTrm*)↑ **do**
    begin *Out_XElStart*(*TermName*[*wsFraenkelTerm*]); *Out_PosAsAttrs*(*nTermPos*); *Out_XAttrEnd*;
    for $i \leftarrow 0$ **to** *nPostqualification*↑.*Count* $-$ 1 **do** *Out_VariableSegment*(*nPostqualification*↑.*Items*↑[i]);
    Out_Term(*nSample*); *Out_Formula*(*nFormula*); *Out_XElEnd*(*TermName*[*wsFraenkelTerm*]);
    end

This code is used in section 1045.

1054. Terms: Simple Fraenkel expressions.

```
Term |= element Simple-Fraenkel-Term {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Variable-Segment*,
  Term
}
```

⟨ Emit XML for simple Fraenkel term (WSM) 1054 ⟩ ≡
  with *SimpleFraenkelTermPtr*(*aTrm*)↑ **do**
    begin *Out_XElStart*(*TermName*[*wsSimpleFraenkelTerm*]); *Out_PosAsAttrs*(*nTermPos*);
    Out_XAttrEnd;
    for $i \leftarrow 0$ **to** *nPostqualification*↑.*Count* $-$ 1 **do** *Out_VariableSegment*(*nPostqualification*↑.*Items*↑[i]);
    Out_Term(*nSample*); *Out_XElEnd*(*TermName*[*wsSimpleFraenkelTerm*]);
    end

This code is used in section 1045.

1055. Terms: qualification.

```
Term |= element Qualification-Term {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term,
  Type
}
```

⟨ Emit XML for qualification term (WSM) 1055 ⟩ ≡
  with *QualifiedTermPtr*(*aTrm*)↑ **do**
    begin *Out_XElStart*(*TermName*[*wsQualificationTerm*]); *Out_PosAsAttrs*(*nTermPos*); *Out_XAttrEnd*;
    Out_Term(*nSubject*); *Out_Type*(*nQualification*); *Out_XElEnd*(*TermName*[*wsQualificationTerm*]);
    end

This code is used in section 1045.

1056. Terms: exactly qualified.

```
Term |= element Exactly-Qualification-Term {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term
}
```

⟨ Emit XML for exactly qualification term (WSM) 1056 ⟩ ≡
  with *ExactlyTermPtr*(*aTrm*)↑ **do**
    begin *Out_XElStart*(*TermName*[*wsQualificationTerm*]); *Out_PosAsAttrs*(*nTermPos*);
    Out_XAttrEnd; *Out_Term*(*nSubject*); *Out_XElEnd*(*TermName*[*wsQualificationTerm*]);
    end

This code is used in section 1045.

1057. Terms: global choice expressions.

```
Term |= element Global-Choice-Term {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Type
}
```

⟨ Emit XML for global choice term (WSM) 1057 ⟩ ≡
  begin *Out_XElStart*(*TermName*[*wsGlobalChoiceTerm*]); *Out_PosAsAttrs*(*aTrm*↑.*nTermPos*);
  Out_XAttrEnd; *Out_Type*(*ChoiceTermPtr*(*aTrm*)↑.*nChoiceType*);
  Out_XElEnd(*TermName*[*wsGlobalChoiceTerm*]);
  end

This code is used in section 1045.

Subsection 21.11.4. Emitting XML for text items

1058. Type-lists are needed for text items.

```
Type-List = element Type-List {
  Type*
}
```

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *OutWSMizFileObj*.*Out_TypeList*(*aTypeList* : *PList*);
  var *i*: *integer*;
  begin *Out_XElStart0*(*XMLElemName*[*elTypeList*]);
  for $i \leftarrow 0$ **to** *aTypeList*↑.*Count* − 1 **do** *Out_Type*(*aTypeList*↑.*Items*↑[*i*]);
  Out_XElEnd(*XMLElemName*[*elTypeList*]);
  end;

1059. Locus.

```
Locus = element Locus {
  attribute idnr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer }
}
```

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
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;

1060.

```
Loci = element Loci { Locus* }
```

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *OutWSMizFileObj.Out_Loci*(*aLoci* : *PList*);
  var *i*: *integer*;
  begin if $(aLoci = \mathbf{nil}) \lor (aLoci\uparrow.Count = 0)$ **then** *Out_XEl1*(*XMLElemName*[*elLoci*])
  else begin *Out_XElStart0*(*XMLElemName*[*elLoci*]);
    for $i \leftarrow 0$ **to** $aLoci\uparrow.Count - 1$ **do** *Out_Locus*(*aLoci*↑.*Items*↑[*i*]);
    Out_XElEnd(*XMLElemName*[*elLoci*]);
    end;
  end;

1061. Patterns.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *OutWSMizFileObj.Out_Pattern*(*aPattern* : *PatternPtr*);
  begin case *aPattern*↑.*nPatternSort* **of**
  itDefPred: ⟨ Emit XML for predicate pattern (WSM) 1062 ⟩;
  itDefFunc: **begin case** *FunctorPatternPtr*(*aPattern*)↑.*nFunctKind* **of**
    InfixFunctor: ⟨ Emit XML for infix functor pattern (WSM) 1063 ⟩;
    CircumfixFunctor: ⟨ Emit XML for bracket functor pattern (WSM) 1064 ⟩;
    endcases;
    end;
  itDefMode: ⟨ Emit XML for mode pattern (WSM) 1065 ⟩;
  end;
itDefAttr: ⟨ Emit XML for attribute pattern (WSM) 1066 ⟩;
  endcases;
  end ;

1062.

```
Predicate-Pattern = element Predicate-Pattern {
  attribute nr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Loci,
  Loci
}
```

⟨ Emit XML for predicate pattern (WSM) 1062 ⟩ ≡
  with *PredicatePatternPtr*(*aPattern*)↑ **do**
    begin *Out_XElStart*(*DefPatternName*[*itDefPred*]);
    Out_XIntAttr(*XMLAttrName*[*atNr*], *nPredSymbol*);
    if *nMizarAppearance* **then** *Out_XAttr*(*XMLAttrName*[*atSpelling*], *PredicateName*[*nPredSymbol*]);
    Out_PosAsAttrs(*nPatternPos*); *Out_XAttrEnd*; *Out_Loci*(*nLeftArgs*); *Out_Loci*(*nRightArgs*);
    Out_XElEnd(*DefPatternName*[*itDefPred*]);
    end

This code is used in section 1061.

1063.

```
Operation-Functor-Pattern = element Operation-Functor-Pattern {
  attribute nr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Loci,
  Loci
}
```

⟨ Emit XML for infix functor pattern (WSM) 1063 ⟩ ≡
  with *InfixFunctorPatternPtr*(*aPattern*)↑ **do**
    begin *Out_XElStart*(*FunctorPatternName*[*InfixFunctor*]);
    Out_XIntAttr(*XMLAttrName*[*atNr*], *nOperSymb*);
    if *nMizarAppearance* **then** *Out_XAttr*(*XMLAttrName*[*atSpelling*], *FunctorName*[*nOperSymb*]);
    Out_PosAsAttrs(*nPatternPos*); *Out_XAttrEnd*; *Out_Loci*(*nLeftArgs*); *Out_Loci*(*nRightArgs*);
    Out_XElEnd(*FunctorPatternName*[*InfixFunctor*]);
    end

This code is used in section 1061.

1064.

```
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
}
```

⟨ Emit XML for bracket functor pattern (WSM) 1064 ⟩ ≡
  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*[*atSpelling*], *RightBracketName*[*nRightBracketSymb*]);
    Out_XAttrEnd; *Out_XElEnd*(*XMLElemName*[*elRightCircumflexSymbol*]); *Out_Loci*(*nArgs*);
    Out_XElEnd(*FunctorPatternName*[*CircumfixFunctor*]);
    end

This code is used in section 1061.

1065.

```
Mode-Pattern = element Mode-Pattern {
  attribute nr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Loci
}
```

⟨ Emit XML for mode pattern (WSM) 1065 ⟩ ≡
  with *ModePatternPtr*(*aPattern*)↑ **do**
    begin *Out_XElStart*(*DefPatternName*[*itDefMode*]);
    Out_XIntAttr(*XMLAttrName*[*atNr*], *nModeSymbol*);
    if *nMizarAppearance* **then** *Out_XAttr*(*XMLAttrName*[*atSpelling*], *ModeName*[*nModeSymbol*]);
    Out_PosAsAttrs(*nPatternPos*); *Out_XAttrEnd*; *Out_Loci*(*nArgs*);
    Out_XElEnd(*DefPatternName*[*itDefMode*])

This code is used in section 1061.

1066. I am confused why there is both a locus and loci elements in an attribute pattern.

```
Attribute-Pattern = element Attribute-Pattern {
  attribute nr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Locus,
  Loci
}
```

⟨ Emit XML for attribute pattern (WSM) 1066 ⟩ ≡
  with *AttributePatternPtr*(*aPattern*)↑ **do**
    begin *Out_XElStart*(*DefPatternName*[*itDefAttr*]); *Out_XIntAttr*(*XMLAttrName*[*atNr*], *nAttrSymbol*);
    if *nMizarAppearance* **then** *Out_XAttr*(*XMLAttrName*[*atSpelling*], *AttributeName*[*nAttrSymbol*]);
    Out_PosAsAttrs(*nPatternPos*); *Out_XAttrEnd*; *Out_Locus*(*nArg*); *Out_Loci*(*nArgs*);
    Out_XElEnd(*DefPatternName*[*itDefAttr*]);
    end

This code is used in section 1061.

1067.

```
Label = element Label {
  attribute idnr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Locus,
  Loci
}
```

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *OutWSMizFileObj.Out_Label*(*aLab* : *LabelPtr*);
    begin
    if (*aLab* ≠ **nil**) { ∧(*aLab.nLabelIdNr* > 0) }
    then
    begin *Out_XElStart*(*XMLElemName*[*elLabel*]);
    Out_XIntAttr(*XMLAttrName*[*atIdNr*], *aLab*↑.*nLabelIdNr*);
    if *nMizarAppearance* **then** *Out_XAttr*(*XMLAttrName*[*atSpelling*], *IdentRepr*(*aLab*↑.*nLabelIdNr*));
    Out_PosAsAttrs(*aLab*↑.*nLabelPos*); *Out_XElEnd0*
    end;
    end ;

1068. Emitting XML for definiens.

```
Definiens = element Definiens {
  attribute kind { "Simple-Definiens" },
  attribute shape { text }?,
  Label,
  (Term | Formula)
} | element Definiens {
  attribute kind { "Conditional-Definiens" },
  attribute shape { text }?,
  Label,
  element Partial-Definiens { (Term | Formula)* },
  (Term | Formula)?
}
```

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *OutWSMizFileObj.Out_Definiens*(*aDef* : *DefiniensPtr*);
  var *i*: *integer*; *lExprKind*: *ExpKind*;
  begin if *aDef* ≠ **nil then**
    with *DefiniensPtr*(*aDef*)↑ **do**
      begin *Out_XElStart*(*XMLElemName*[*elDefiniens*]); *Out_PosAsAttrs*(*nDefPos*);
      case *nDefSort* **of**
      SimpleDefiniens: **with** *SimpleDefiniensPtr*(*aDef*)↑, *nExpression*↑ **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;
      ConditionalDefiniens: **with** *ConditionalDefiniensPtr*(*aDef*)↑ **do**
          begin *Out_XAttr*(*XMLAttrName*[*atKind*], *DefiniensKindName*[*ConditionalDefiniens*]);
          lExprKind ← *exFormula*;
          if *nOtherwise* ≠ **nil then** *lExprKind* ← *nOtherwise*↑.*nExprKind*
          else if *nConditionalDefiniensList*↑.*Count* > 0 **then** *lExprKind* ←
                PartDefPtr(*nConditionalDefiniensList*↑.*Items*↑[0])↑.*nPartDefiniens*↑.*nExprKind*;
          Out_XAttr(*XMLAttrName*[*atShape*], *ExpName*[*lExprKind*]); *Out_XAttrEnd*;
          Out_Label(*nDefLabel*);
          for *i* ← 0 **to** *nConditionalDefiniensList*↑.*Count* − 1 **do**
            with *PartDefPtr*(*nConditionalDefiniensList*↑.*Items*↑[*I*])↑ **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*]);
              end;
          if *nOtherwise* ≠ **nil then**
            with *nOtherwise*↑ **do**
              case *nExprKind* **of**
              exTerm: *Out_Term*(*TermPtr*(*nExpr*));
              exFormula: *Out_Formula*(*FormulaPtr*(*nExpr*));

```
          endcases;
        end;
      endcases; Out_XElEnd(XMLElemName[elDefiniens]);
      end;
  end;
```

1069.

```
Proposition = element Proposition {
  Label,
  Formula
}
```

⟨Implementation for wsmarticle.pas 854⟩ +≡

```
procedure OutWSMizFileObj.Out_Proposition(aProp : PropositionPtr);
  begin Out_XElStart(XMLElemName[elProposition]); Out_XAttrEnd; Out_Label(aProp↑.nLab);
  Out_Formula(aProp↑.nSentence); Out_XElEnd(XMLElemName[elProposition]);
  end;
```

1070.

```
Local-Reference = element Local-Reference {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  attribute idnr { xsd:integer },
  attribute spelling { text }?
}
```

⟨Implementation for wsmarticle.pas 854⟩ +≡

```
procedure OutWSMizFileObj.Out_LocalReference(aRef : LocalReferencePtr);
  begin with LocalReferencePtr(aRef)↑ do
    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;
```

1071.

```
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 }
})*
```

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
procedure *OutWSMizFileObj.Out_References*(*aRefs* : *PList*);
  var *i*: *integer*;
  begin for $i \leftarrow 0$ **to** *aRefs*↑.*Count* − 1 **do**
    with *ReferencePtr*(*aRefs*↑.*Items*↑[*i*])↑ **do**
      case *nRefSort* **of**
      LocalReference: *Out_LocalReference*(*aRefs*↑.*Items*↑[*i*]);
      TheoremReference: **begin** *Out_XElStart*(*ReferenceKindName*[*TheoremReference*]);
        Out_PosAsAttrs(*nRefPos*);
        Out_XIntAttr(*XMLAttrName*[*atNr*], *TheoremReferencePtr*(*aRefs*↑.*Items*↑[*i*])↑.*nArticleNr*);
        if *nMizarAppearance* **then** *Out_XAttr*(*XMLAttrName*[*atSpelling*],
            MMLIdentifierName[*TheoremReferencePtr*(*aRefs*↑.*Items*↑[*i*])↑.*nArticleNr*]);
        Out_XIntAttr(*XMLAttrName*[*atNumber*], *TheoremReferencePtr*(*aRefs*↑.*Items*↑[*i*])↑.*nTheoNr*);
        Out_XElEnd0;
        end;
      DefinitionReference: **begin** *Out_XElStart*(*ReferenceKindName*[*DefinitionReference*]);
        Out_PosAsAttrs(*nRefPos*);
        Out_XIntAttr(*XMLAttrName*[*atNr*], *DefinitionReferencePtr*(*aRefs*↑.*Items*↑[*i*])↑.*nArticleNr*);
        if *nMizarAppearance* **then** *Out_XAttr*(*XMLAttrName*[*atSpelling*],
            MMLIdentifierName[*TheoremReferencePtr*(*aRefs*↑.*Items*↑[*i*])↑.*nArticleNr*]);
        Out_XIntAttr(*XMLAttrName*[*atNumber*], *DefinitionReferencePtr*(*aRefs*↑.*Items*↑[*i*])↑.*nDefNr*);
        Out_XElEnd0;
        end;
      endcases;
  end;

1072.

```
Link = element Link {
  attribute col { xsd:integer },
  attribute line { xsd:integer }
}
```

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *OutWSMizFileObj.Out_Link*(*aInf* : *JustificationPtr*);
  begin with *StraightforwardJustificationPtr*(*aInf*)↑ **do**
    if *nLinked* **then**
      begin *Out_XElStart*(*XMLElemName*[*elLink*]); *Out_PosAsAttrs*(*nLinkPos*); *Out_XElEnd0*;
      end;
  end;

1073.

```
Scheme-Justification = element Scheme-Justification {
  attribute nr { xsd:integer },
  attribute idnr { xsd:integer },
  attribute spelling { text }?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  attribute poscol { xsd:integer },
  attribute posline { xsd:integer },
  References
}
```

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
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;

1074.

```
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
)
```

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
procedure *OutWSMizFileObj.Out_Justification*(*aInf* : *JustificationPtr*; *aBlock* : *wsBlockPtr*);
  begin case *aInf*↑.*nInfSort* **of**
  infStraightforwardJustification: **with** *StraightforwardJustificationPtr*(*aInf*)↑ **do**
    begin *Out_XElStart*(*InferenceName*[*infStraightforwardJustification*]); *Out_PosAsAttrs*(*nInfPos*);
    if $\neg$*nLinked* $\wedge$ (*nReferences*↑.*Count* $= 0$) **then** *Out_XElEnd0*
    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*↑.*nInfPos*);
  infSkippedProof: *Out_XElWithPos*(*InferenceName*[*infSkippedProof*], *aInf*↑.*nInfPos*);
  infProof: *Out_Block*(*aBlock*);
  endcases;
  end;

1075.

```
Compact-Statement = (Proposition, Justification)
```

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
procedure *OutWSMizFileObj.Out_CompactStatement*(*aCStm* : *CompactStatementPtr*;
    aBlock : *wsBlockPtr*);
  begin with *aCStm*↑ **do**
    begin *Out_Proposition*(*nProp*); *Out_Justification*(*nJustification*, *aBlock*);
    end;
  end;

1076.

```
Regular-Statement =
( (Label, Block)
| Compact-Statement
| (Compact-Statement,
   element Iterative-Step {
     attribute col { xsd:integer },
     attribute line { xsd:integer },
     Term,
     Justification
   })*
)
```

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

procedure *OutWSMizFileObj.Out_RegularStatement*(*aRStm* : *RegularStatementPtr*; *aBlock* : *wsBlockPtr*);
  var *i*: *integer*;
  begin case *aRStm*↑.*nStatementSort* **of**
  stDiffuseStatement: **begin** *Out_Label*(*DiffuseStatementPtr*(*aRStm*)↑.*nLab*); *Out_Block*(*aBlock*);
    end;
  stCompactStatement: *Out_CompactStatement*(*CompactStatementPtr*(*aRStm*), *aBlock*);
  stIterativeEquality: **begin** *Out_CompactStatement*(*CompactStatementPtr*(*aRStm*), **nil**);
    with *IterativeEqualityPtr*(*aRStm*)↑ **do**
      for $i \leftarrow 0$ **to** *nIterSteps*↑.*Count* − 1 **do**
        with *IterativeStepPtr*(*nIterSteps*↑.*Items*↑[*i*])↑ **do**
          begin *Out_XElStart*(*XMLElemName*[*elIterativeStep*]); *Out_PosAsAttrs*(*nIterPos*);
          Out_XAttrEnd; *Out_Term*(*nTerm*); *Out_Justification*(*nJustification*, **nil**);
          Out_XElEnd(*XMLElemName*[*elIterativeStep*]);
          end;
    end;
  endcases;
  end;

1077.

```
Variables = element Variables {
  Variable*
}
```

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

procedure *OutWSMizFileObj.Out_ReservationSegment*(*aRes* : *ReservationSegmentPtr*);
  var *i*: *integer*;
  begin with *aRes*↑ **do**
    begin *Out_XElStart0*(*XMLElemName*[*elVariables*]);
    for $i \leftarrow 0$ **to** *nIdentifiers*↑.*Count* − 1 **do** *Out_ReservedVariable*(*nIdentifiers*↑.*Items*↑[*i*]);
    Out_XElEnd(*XMLElemName*[*elVariables*]); *Out_Type*(*nResType*);
    end;
  end;

1078. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

procedure *OutWSMizFileObj.Out_SchemeNameInSchemeHead*(*aSch* : *SchemePtr*);
  begin *Out_XIntAttr*(*XMLAttrName*[*atIdNr*], *aSch*↑.*nSchemeIdNr*);
  if *nMizarAppearance* **then** *Out_XAttr*(*XMLAttrName*[*atSpelling*], *IdentRepr*(*aSch*↑.*nSchemeIdNr*));
  end;

1079. ⟨Implementation for `wsmarticle.pas` 854⟩ +≡

procedure *OutWSMizFileObj.Out_ItemContentsAttr*(*aWSItem* : *WSItemPtr*);
  begin with *aWSItem*↑ **do**
    begin *CurPos* ← *nItemPos*;
    if *nDisplayInformationOnScreen* **then** *DisplayLine*(*CurPos.Line*, *ErrorNbr*);
    case *nItemKind* **of**
    itDefinition, *itSchemeBlock*, *itSchemeHead*, *itTheorem*, *itAxiom*, *itReservation*: ;
    itSection: ;
    itConclusion, *itRegularStatement*: **case** *RegularStatementPtr*(*nContent*)↑.*nStatementSort* **of**
      stDiffuseStatement:
          Out_XAttr(*XMLAttrName*[*atShape*], *RegularStatementName*[*stDiffuseStatement*]);
      stCompactStatement:
          Out_XAttr(*XMLAttrName*[*atShape*], *RegularStatementName*[*stCompactStatement*]);
      stIterativeEquality: *Out_XAttr*(*XMLAttrName*[*atShape*], *RegularStatementName*[*stIterativeEquality*]);
      endcases;
    itChoice, *itReconsider*, *itPrivFuncDefinition*, *itPrivPredDefinition*, *itConstantDefinition*, *itGeneralization*,
        itLociDeclaration, *itExistentialAssumption*, *itExemplification*, *itPerCases*, *itCaseBlock*: ;
    itCaseHead, *itSupposeHead*, *itAssumption*: ;
    itCorrCond: *Out_XAttr*(*XMLAttrName*[*atCondition*],
        CorrectnessName[*CorrectnessConditionPtr*(*nContent*)↑.*nCorrCondSort*]);
    itCorrectness: *Out_XAttr*(*XMLAttrName*[*atCondition*], *CorrectnessName*[*syCorrectness*]);
    itProperty:
        Out_XAttr(*XMLAttrName*[*atProperty*], *PropertyName*[*PropertyPtr*(*nContent*)↑.*nPropertySort*]);
    itDefFunc: *Out_XAttr*(*XMLAttrName*[*atShape*],
        DefiningWayName[*FunctorDefinitionPtr*(*nContent*)↑.*nDefiningWay*]);
    itDefPred, *itDefMode*, *itDefAttr*, *itDefStruct*, *itPredSynonym*, *itPredAntonym*, *itFuncNotation*,
        itModeNotation, *itAttrSynonym*, *itAttrAntonym*, *itCluster*, *itIdentify*, *itReduction*: ;
    itPropertyRegistration:
        Out_XAttr(*XMLAttrName*[*atProperty*], *PropertyName*[*PropertyPtr*(*nContent*)↑.*nPropertySort*]);
    itPragma:
        Out_XAttr(*XMLAttrName*[*atSpelling*], *QuoteStrForXML*(*PragmaPtr*(*nContent*)↑.*nPragmaStr*));
    endcases;
    end;
  end;

1080. Emitting XML for item contents. This is used to expedite emitting the XML for a text-item (§1095).

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *OutWSMizFileObj.Out_ItemContents*(*aWSItem* : *WSItemPtr*);
  var *i*, *j*: *integer*; *s*: *CorrectnessKind*;
  begin with *aWSItem*↑ **do**
    begin case *nItemKind* **of**
    itDefinition: *Out_Block*(*nBlock*);
    itSchemeBlock: *Out_Block*(*nBlock*);
    itSchemeHead: ⟨ Emit XML for schema (WSM) 1081 ⟩;
    itTheorem: *Out_CompactStatement*(*CompactStatementPtr*(*nContent*), *nBlock*);
    itAxiom: **begin end**;
    itReservation: *Out_ReservationSegment*(*ReservationSegmentPtr*(*nContent*));
    itSection: ;
    itConclusion, *itRegularStatement*: *Out_RegularStatement*(*RegularStatementPtr*(*nContent*), *nBlock*);
    itChoice: ⟨ Emit XML for `consider` contents (WSM) 1082 ⟩;
    itReconsider: ⟨ Emit XML for `reconsider` contents (WSM) 1083 ⟩;
    ⟨ Emit XML for definition-related items (WSM) 1084 ⟩;
    itPredSynonym, *itPredAntonym*, *itFuncNotation*, *itModeNotation*, *itAttrSynonym*, *itAttrAntonym*:
        with *NotationDeclarationPtr*(*nContent*)↑ **do**
      begin *Out_Pattern*(*nOriginPattern*); *Out_Pattern*(*nNewPattern*);
      end;
    ⟨ Emit XML for registration-related items (WSM) 1093 ⟩;
    itPragma: ;
    itIncorrItem: ;
    end;
    endcases;
  end;

1081.

```
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* },
      Type
    } | 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* }?
}
```

⟨ Emit XML for schema (WSM) 1081 ⟩ ≡

with *SchemePtr*(*nContent*)↑ **do**
  begin *Out_XElStart*(*XMLElemName*[*elScheme*]);
  Out_SchemeNameInSchemeHead(*SchemePtr*(*nContent*)); *Out_XElEnd0*;
  Out_XElStart0(*XMLElemName*[*elSchematicVariables*]);
  for $j \leftarrow 0$ **to** *nSchemeParams*↑.*Count* − 1 **do**
    case *SchemeSegmentPtr*(*nSchemeParams*.*Items*↑[*j*])↑.*nSegmSort* **of**
    PredicateSegment: **with** *PredicateSegmentPtr*(*nSchemeParams*.*Items*↑[*j*])↑ **do**
        begin *Out_XElStart*(*SchemeSegmentName*[*PredicateSegment*]); *Out_PosAsAttrs*(*nSegmPos*);
        Out_XAttrEnd; *Out_XElStart0*(*XMLElemName*[*elVariables*]);
        for $i \leftarrow 0$ **to** *nVars*↑.*Count* − 1 **do** *Out_Variable*(*nVars*.*Items*↑[*i*]);
        Out_XElEnd(*XMLElemName*[*elVariables*]); *Out_TypeList*(*nTypeExpList*);
        Out_XElEnd(*SchemeSegmentName*[*PredicateSegment*]);
        end;
    FunctorSegment: **with** *FunctorSegmentPtr*(*nSchemeParams*.*Items*↑[*j*])↑ **do**
        begin *Out_XElStart*(*SchemeSegmentName*[*FunctorSegment*]); *Out_PosAsAttrs*(*nSegmPos*);
        Out_XAttrEnd; *Out_XElStart0*(*XMLElemName*[*elVariables*]);
        for $i \leftarrow 0$ **to** *nVars*↑.*Count* − 1 **do** *Out_Variable*(*nVars*.*Items*↑[*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* ≠ **nil**) ∧ (*nSchemePremises*↑.*Count* > 0) **then**
    begin *Out_XElStart0*(*XMLElemName*[*elProvisionalFormulas*]);
    for $i \leftarrow 0$ **to** *nSchemePremises*↑.*Count* − 1 **do** *Out_Proposition*(*nSchemePremises*↑.*Items*↑[*i*]);
    Out_XElEnd(*XMLElemName*[*elProvisionalFormulas*]);
    end;

end

This code is used in section 1080.

1082.

```
Item-contents |= Consider-Statement-contents
Consider-Statement-contents =
( Variable-Segment*,
  element Conditions { Proposition },
  Justification
)
```

⟨ Emit XML for `consider` contents (WSM) 1082 ⟩ ≡
with *ChoiceStatementPtr*(*nContent*)↑ **do**
begin for $i \leftarrow 0$ **to** *nQualVars*↑.*Count* − 1 **do** *Out_VariableSegment*(*nQualVars*↑.*Items*↑[*i*]);
Out_XElStart0(*XMLElemName*[*elConditions*]);
for $i \leftarrow 0$ **to** *nConditions*↑.*Count* − 1 **do** *Out_Proposition*(*nConditions*↑.*Items*↑[*i*]);
Out_XElEnd(*XMLElemName*[*elConditions*]); *Out_Justification*(*nJustification*, **nil**);
end

This code is used in section 1080.

1083.

```
Item-contents |= Type-Changing-Statement-contents
Type-Changing-Statement-contents =
((element Equality {
    Variable,
    Term
  } | Variable),
 Type)
```

⟨ Emit XML for `reconsider` contents (WSM) 1083 ⟩ ≡
with *TypeChangingStatementPtr*(*nContent*)↑ **do**
begin for $i \leftarrow 0$ **to** *nTypeChangeList*↑.*Count* − 1 **do**
case *TypeChangePtr*(*nTypeChangeList*.*Items*↑[*i*])↑.*nTypeChangeKind* **of**
Equating: **begin** *Out_XElStart0*(*XMLElemName*[*elEquality*]);
Out_Variable(*TypeChangePtr*(*nTypeChangeList*.*Items*↑[*i*])↑.*nVar*);
Out_Term(*TypeChangePtr*(*nTypeChangeList*.*Items*↑[*i*])↑.*nTermExpr*);
Out_XElEnd(*XMLElemName*[*elEquality*]);
end;
VariableIdentifier: **begin** *Out_Variable*(*TypeChangePtr*(*nTypeChangeList*.*Items*↑[*i*])↑.*nVar*);
end;
endcases;
Out_Type(*nTypeExpr*); *Out_Justification*(*nJustification*, **nil**);
end

This code is used in section 1080.

1084. We will need to recall *Out_Variable* (§1022) fr *PrivateFunctorDefinitionObj* (§893).

```
Item-contents |=
  (Variable, Type-List, Term) # private functors and predicates
| (Variable, Term)            # constants
| Variable-Segment            # loci
```

⟨ Emit XML for definition-related items (WSM) 1084 ⟩ ≡
itPrivFuncDefinition: **with** *PrivateFunctorDefinitionPtr*(*nContent*)↑ **do**
  begin *Out_Variable*(*nFuncId*); *Out_TypeList*(*nTypeExpList*); *Out_Term*(*nTermExpr*);
  end;
itPrivPredDefinition: **with** *PrivatePredicateDefinitionPtr*(*nContent*)↑ **do**
  begin *Out_Variable*(*nPredId*); *Out_TypeList*(*nTypeExpList*); *Out_Formula*(*nSentence*);
  end;
itConstantDefinition: **with** *ConstantDefinitionPtr*(*nContent*)↑ **do**
  begin *Out_Variable*(*nVarId*); *Out_Term*(*nTermExpr*);
  end;
itLociDeclaration, *itGeneralization*: *Out_VariableSegment*(*QualifiedSegmentPtr*(*nContent*));
itCaseHead, *itSupposeHead*, *itAssumption*: ⟨ Emit XML for assumptions item (WSM) 1092 ⟩;

See also sections 1085, 1086, 1087, 1088, 1089, 1090, and 1091.
This code is used in section 1080.

1085.

```
Item-contents |=
( Variable-Segment*,
  element Conditions { Proposition* } )
```

⟨ Emit XML for definition-related items (WSM) 1084 ⟩ +≡
itExistentialAssumption: **with** *ExistentialAssumptionPtr*(*nContent*)↑ **do**
  begin for $i \leftarrow 0$ **to** *nQVars*↑.*Count* − 1 **do** *Out_VariableSegment*(*nQVars*↑.*Items*↑[*i*]);
  Out_XElStart0(*XMLElemName*[*elConditions*]);
  for $i \leftarrow 0$ **to** *nConditions*↑.*Count* − 1 **do** *Out_Proposition*(*nConditions*↑.*Items*↑[*i*]);
  Out_XElEnd(*XMLElemName*[*elConditions*]);
  end;

1086.

```
Item-contents |= ( Variable?, Term? ) # Exemplification
               | Justification        # percases, correctness-condition
               | Block                # case block
```

⟨ Emit XML for definition-related items (WSM) 1084 ⟩ +≡
itExemplification: **with** *ExamplePtr*(*nContent*)↑ **do**
  begin if *nVarId* ≠ **nil then** *Out_Variable*(*nVarId*);
  if *nTermExpr* ≠ **nil then** *Out_Term*(*nTermExpr*);
  end;
itPerCases: *Out_Justification*(*JustificationPtr*(*nContent*), **nil**);
itCaseBlock: *Out_Block*(*nBlock*);
itCorrCond: *Out_Justification*(*CorrectnessConditionPtr*(*nContent*)↑.*nJustification*, *nBlock*);

1087.

```
Item-contents |=
  element CorrectnessConditions { # sic!
    element Correctness { attribute condition { text } }*,
      Justification }
|Justification # Property
```

⟨ Emit XML for definition-related items (WSM) 1084 ⟩ +≡
itCorrectness: **begin** *Out_XElStart0*(*XMLElemName*[*elCorrectnessConditions*]);
  for $s \in$ *CorrectnessConditionsPtr*(*nContent*)↑.*nConditions* **do**
    begin *Out_XElStart*(*ItemName*[*itCorrectness*]);
    Out_XAttr(*XMLAttrName*[*atCondition*], *CorrectnessName*[s]); *Out_XElEnd0*;
    end;
  Out_XElEnd(*XMLElemName*[*elCorrectnessConditions*]);
  Out_Justification(*CorrectnessPtr*(*nContent*)↑.*nJustification*, *nBlock*);
  end;
itProperty: *Out_Justification*(*PropertyPtr*(*nContent*)↑.*nJustification*, *nBlock*);

1088.

```
Item-contents |=
( element Redefine { }?,
  Pattern,
  element Standard-Mode { Type },
  | element Expandable-Mode {
      element Type-Specification { Type }?,
      Definiens
    })
```

⟨ Emit XML for definition-related items (WSM) 1084 ⟩ +≡
itDefMode: **with** *ModeDefinitionPtr*(*nContent*)↑ **do**
    begin if *nRedefinition* **then** *Out_XEl1*(*XMLElemName*[*elRedefine*]);
    Out_Pattern(*nDefModePattern*);
    case *nDefKind* **of**
    defExpandableMode: **begin** *Out_XElStart0*(*ModeDefinitionSortName*[*defExpandableMode*]);
      Out_Type(*ExpandableModeDefinitionPtr*(*nContent*)↑.*nExpansion*);
      Out_XElEnd(*ModeDefinitionSortName*[*defExpandableMode*]);
      end;
    defStandardMode: **with** *StandardModeDefinitionPtr*(*nContent*)↑ **do**
        begin *Out_XElStart0*(*ModeDefinitionSortName*[*defStandardMode*]);
        if *nSpecification* $\neq$ **nil then**
          begin *Out_XElStart0*(*XMLElemName*[*elTypeSpecification*]); *Out_Type*(*nSpecification*);
          Out_XElEnd(*XMLElemName*[*elTypeSpecification*]);
          end;
        Out_Definiens(*nDefiniens*); *Out_XElEnd*(*ModeDefinitionSortName*[*defStandardMode*]);
        end;
    endcases;
    end;

1089.

```
Item-contents |=
(element Redefine { }?,
 Pattern,
 Definiens)
```

⟨ Emit XML for definition-related items (WSM) 1084 ⟩ +≡
itDefAttr: **with** *AttributeDefinitionPtr*(*nContent*)↑ **do**
  begin if *nRedefinition* **then** *Out_XEl1*(*XMLElemName*[*elRedefine*]);
  Out_Pattern(*nDefAttrPattern*); *Out_Definiens*(*nDefiniens*);
  end;
itDefPred: **with** *PredicateDefinitionPtr*(*nContent*)↑ **do**
  begin if *nRedefinition* **then** *Out_XEl1*(*XMLElemName*[*elRedefine*]);
  Out_Pattern(*nDefPredPattern*); *Out_Definiens*(*nDefiniens*);
  end;

1090.

```
Item-contents |=
(element Redefine { }?,
Pattern,
element Type-Specification { Type }?,
Definiens)
```

⟨ Emit XML for definition-related items (WSM) 1084 ⟩ +≡
itDefFunc: **with** *FunctorDefinitionPtr*(*nContent*)↑ **do**
  begin if *nRedefinition* **then** *Out_XEl1*(*XMLElemName*[*elRedefine*]);
  Out_Pattern(*nDefFuncPattern*);
  if *nSpecification* ≠ **nil then**
    begin *Out_XElStart0*(*XMLElemName*[*elTypeSpecification*]); *Out_Type*(*nSpecification*);
    Out_XElEnd(*XMLElemName*[*elTypeSpecification*]);
    end;
  Out_Definiens(*nDefiniens*);
  end;

1091.

```
Item-contents |=
(element Ancestors { Type* },
   attribute nr { xsd:integer },
   attribute spelling { text }?,
   attribute col { xsd:integer },
   attribute line { xsd:integer },
   Loci,
   (element Field-Segment {
      attribute col { xsd:integer },
      attribute line { xsd:integer },
      (element Selector {
        attribute nr { xsd:integer },
        attribute spelling { text }?,
        attribute col { xsd:integer },
        attribute line { xsd:integer }
      })*,
      Type
   )*
 },
```

⟨ Emit XML for definition-related items (WSM) 1084 ⟩ +≡
itDefStruct: **with** *StructureDefinitionPtr*(*nContent*)↑ **do**
  begin *Out_XElStart0*(*XMLElemName*[*elAncestors*]);
  for $i \leftarrow 0$ **to** *nAncestors*↑.*Count* $- 1$ **do** *Out_Type*(*nAncestors*↑.*Items*↑[*i*]);
  Out_XElEnd(*XMLElemName*[*elAncestors*]); *Out_XElStart*(*DefPatternName*[*itDefStruct*]);
  Out_XIntAttr(*XMLAttrName*[*atNr*], *nDefStructPattern*↑.*nModeSymbol*);
  if *nMizarAppearance* **then**
   Out_XAttr(*XMLAttrName*[*atSpelling*], *StructureName*[*nDefStructPattern*↑.*nModeSymbol*]);
  Out_PosAsAttrs(*nStrPos*); *Out_XAttrEnd*; *Out_Loci*(*nDefStructPattern*↑.*nArgs*);
  for $i \leftarrow 0$ **to** *nSgmFields*↑.*Count* $- 1$ **do**
   with *FieldSegmentPtr*(*nSgmFields*↑.*Items*↑[*i*])↑ **do**
    begin *Out_XElStart*(*XMLElemName*[*elFieldSegment*]); *Out_PosAsAttrs*(*nFieldSegmPos*);
    Out_XAttrEnd;
    for $j \leftarrow 0$ **to** *nFields*↑.*Count* $- 1$ **do**
     with *FieldSymbolPtr*(*nFields*↑.*Items*↑[*j*])↑ **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*
      end;
    Out_Type(*nSpecification*); *Out_XElEnd*(*XMLElemName*[*elFieldSegment*]);
    end;
  Out_XElEnd(*DefPatternName*[*itDefStruct*]);
  end

1092.

```
Item-contents |= (element Single-Assumption {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Proposition
} | element Collective-Assumption {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  element Conditions { Proposition* }
})
```

⟨Emit XML for assumptions item (WSM) 1092⟩ ≡
  case *AssumptionPtr*(*nContent*)↑.*nAssumptionSort* **of**
  SingleAssumption: **begin** *Out_XElStart*(*AssumptionKindName*[*SingleAssumption*]);
    Out_PosAsAttrs(*AssumptionPtr*(*nContent*)↑.*nAssumptionPos*); *Out_XAttrEnd*;
    Out_Proposition(*SingleAssumptionPtr*(*nContent*)↑.*nProp*);
    Out_XElEnd(*AssumptionKindName*[*SingleAssumption*]);
    end;
  CollectiveAssumption: **begin** *Out_XElStart*(*AssumptionKindName*[*CollectiveAssumption*]);
    Out_PosAsAttrs(*AssumptionPtr*(*nContent*)↑.*nAssumptionPos*); *Out_XAttrEnd*;
    Out_XElStart0(*XMLElemName*[*elConditions*]);
    with *CollectiveAssumptionPtr*(*nContent*)↑ **do**
      for $i \leftarrow 0$ **to** *nConditions*↑.*Count* $- 1$ **do** *Out_Proposition*(*nConditions*↑.*Items*↑[*i*]);
    Out_XElEnd(*XMLElemName*[*elConditions*]);
    Out_XElEnd(*AssumptionKindName*[*CollectiveAssumption*]);
    end;
  endcases

This code is used in section 1084.

1093. We have cluster registrations and non-cluster registrations.

```
Existential-Registration-content = element Existential-Registration {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Adjective-Cluster,
  Type
}
Conditional-Registration-content = element Conditional-Registration {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Adjective-Cluster, Adjective-Cluster,
  Type
}
Functorial-Registration-content = element Functorial-Registration {
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  Term,
  Adjective-Cluster,
  Type?
}
```

⟨ Emit XML for registration-related items (WSM) 1093 ⟩ ≡
itCluster: **case** *ClusterPtr*(*nContent*)↑.*nClusterKind* **of**
  ExistentialRegistration: **with** *EClusterPtr*(*nContent*)↑ **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*)↑ **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*)↑ **do**
    begin *Out_XElStart*(*ClusterRegistrationName*[*FunctorialRegistration*]);
    Out_PosAsAttrs(*nClusterPos*); *Out_XAttrEnd*; *Out_Term*(*nClusterTerm*);
    Out_AdjectiveList(*nConsequent*);
    if *nClusterType* ≠ **nil then** *Out_Type*(*nClusterType*);
    Out_XElEnd(*ClusterRegistrationName*[*FunctorialRegistration*]);
    end;
  endcases;

See also section 1094.

This code is used in section 1080.

1094.

```
Identify-Registration-content =
(Pattern, Pattern,
  element LociEquality {
      attribute col { xsd:integer },
      attribute line { xsd:integer },
      Locus, Locus
    }*
  })
Sethood-Registration-content = (Type, Justification)
Reduction-Registration-content = (Term, Term)
```

⟨ Emit XML for registration-related items (WSM) 1093 ⟩ +≡
itIdentify: **with** *IdentifyRegistrationPtr*(*nContent*)↑ **do**
  begin *Out_Pattern*(*nOriginPattern*); *Out_Pattern*(*nNewPattern*);
  if *nEqLociList* $\neq$ **nil then**
    begin for $i \leftarrow 0$ **to** *nEqLociList*↑.*Count* $- 1$ **do**
      with *LociEqualityPtr*(*nEqLociList*↑.*Items*↑[i])↑ **do**
        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*)↑.*nPropertySort* **of**
 sySethood: **with** *SethoodRegistrationPtr*(*nContent*)↑ **do**
    begin *Out_Type*(*nSethoodType*); *Out_Justification*(*nJustification*, *nBlock*);
    end;
 endcases;
itReduction: **with** *ReduceRegistrationPtr*(*nContent*)↑ **do**
  begin *Out_Term*(*nOriginTerm*); *Out_Term*(*nNewTerm*);
  end

1095. Emitting an item.

```
Item = element Item {
  attribute kind { text },
  Item-contents-attribute?,
  attribute col { xsd:integer },
  attribute line { xsd:integer },
  attribute posline { xsd:integer },
  attribute poscol { xsd:integer },
  (Block | Item-contents)?
}
```

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

procedure *OutWSMizFileObj.Out_Item*(*aWSItem* : *WSItemPtr*);
  var *i*, *j*: *integer*;
  begin with *aWSItem*↑ **do**
    begin *CurPos* ← *nItemPos*; *Out_XElStart*(*XMLElemName*[*elItem*]);
    Out_XAttr(*XMLAttrName*[*atKind*], *ItemName*[*nItemKind*]);
    if *nContent* ≠ **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* ≠ **nil then** *Out_Block*(*nBlock*);
      end
    else *Out_ItemContents*(*aWsItem*);
    Out_XElEnd(*XMLElemName*[*elItem*]);
    end;
  end;

1096. Writing out to an XML file.

procedure *Write_WSMizArticle*(*aWSTextProper* : *wsTextProperPtr*; *aFileName* : *string*);
  var *lWSMizOutput*: *OutWSMizFilePtr*;
  begin *InitScannerNames*; *lWSMizOutput* ← *new*(*OutWSMizFilePtr*, *OpenFile*(*aFileName*));
  lWSMizOutput↑.*nMizarAppearance* ← *true*; *lWSMizOutput*↑.*Out_TextProper*(*aWSTextProper*);
  dispose(*lWSMizOutput*, *Done*);
  end;

Section 21.12. READING WSM FILES (DEFERRED)

1097. Reading a WSM file amounts to reading an XML file, which means that the *XMLInStream* class (§528) is a natural parent class. Recall, the state of the *XMLInStream* contains the current start tag and a dictionary for the attributes and their values.

The code is a "mirror image" to writing XML files, and the XML schema guides the implementation.

⟨Publicly declared types in `wsmarticle.pas` 852⟩ +≡
InWSMizFilePtr = ↑*InWSMizFileObj*;
InWSMizFileObj = **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_ImplicitlyQualifiedSegment*: *ImplicitlyQualifiedSegmentPtr*; *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*;
  function *Read_LocalReference*: *LocalReferencePtr*; *virtual*;
  function *Read_References*: *PList*; *virtual*;
  function *Read_StraightforwardJustification*: *StraightforwardJustificationPtr*; *virtual*;
  function *Read_SchemeJustification*: *SchemeJustificationPtr*; *virtual*;
  function *Read_Justification*: *JustificationPtr*; *virtual*;
  function *Read_RegularStatement*(**const** *aShape*: *string*): *RegularStatementPtr*; *virtual*;
end ;

1098. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *InWSMizFileObj.OpenFile*(**const** *aFileName*: *string*);
  begin *inherited OpenFile*(*aFileName*); *nDisplayInformationOnScreen* ← *false*;
  end;
destructor *InWSMizFileObj.Done*;
  begin *inherited Done*;
  end;

1099. Getting the value for an attribute. Returns **nil** if there is no attribute with the given name. (Recall (§490), an *XMLAttr* is just a wrapper around a string *nValue*.)

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.GetAttrValue*(**const** *aAttrName*: *string*): *string*;
  var *lObj*: *PObject*;
  begin *result* ← ´´; *lObj* ← *nAttrVals.ObjectOf*(*aAttrName*);
  if *lObj* ≠ **nil then** *result* ← *XMLAttrPtr*(*lObj*)↑.*nValue*;
  end;

1100. We can query for the *position* of the XML attribute.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.GetAttrPos*: *Position*;
  var *lLine*, *lCol*: *XMLAttrPtr*; *lCode*: *integer*;
  begin *result.Line* ← 1; *result.Col* ← 1;
  lLine ← *XMLAttrPtr*(*nAttrVals.ObjectOf*(*XMLAttrName*[*atLine*]));
  lCol ← *XMLAttrPtr*(*nAttrVals.ObjectOf*(*XMLAttrName*[*atCol*]));
  if (*lLine* ≠ **nil**) ∧ (*lCol* ≠ **nil**) **then**
    begin *Val*(*lLine*↑.*nValue*, *result.Line*, *lCode*); *Val*(*lCol*↑.*nValue*, *result.Col*, *lCode*);
    end;
  end;

1101. The state of the WSM parser may be described with a handful of lookup tables.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
var *ElemLookupTable*, *AttrLookupTable*, *BlockLookUpTable*, *ItemLookUpTable*, *FormulaKindLookupTable*,
    TermKindLookupTable, *PatternKindLookupTable*, *CorrectnessKindLookupTable*,
    PropertyKindLookupTable: *MSortedStrList*;

procedure *InitWSLookupTables*;
  var *e*: *XMLElemKind*; *a*: *XMLAttrKind*; *b*: *BlockKind*; *i*: *ItemKind*; *f*: *FormulaSort*; *t*: *TermSort*;
    p: *PropertyKind*; *c*: *CorrectnessKind*;
  begin *ElemLookupTable*.*Init*(*Ord*(*High*(*XMLElemKind*)) + 1);
  AttrLookupTable.*Init*(*Ord*(*High*(*XMLAttrKind*)) + 1);
  BlockLookupTable.*Init*(*Ord*(*High*(*BlockKind*)) + 1); *ItemLookupTable*.*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* ← *Low*(*XMLElemKind*) **to** *High*(*XMLElemKind*) **do**
    ElemLookupTable.*Insert*(*new*(*MStrPtr*, *Init*(*XMLElemName*[*e*])));
  for *a* ← *Low*(*XMLAttrKind*) **to** *High*(*XMLAttrKind*) **do**
    AttrLookupTable.*Insert*(*new*(*MStrPtr*, *Init*(*XMLAttrName*[*a*])));
  for *b* ← *Low*(*BlockKind*) **to** *High*(*BlockKind*) **do**
    BlockLookupTable.*Insert*(*new*(*MStrPtr*, *Init*(*BlockName*[*b*])));
  for *i* ← *Low*(*ItemKind*) **to** *High*(*ItemKind*) **do**
    ItemLookupTable.*Insert*(*new*(*MStrPtr*, *Init*(*ItemName*[*i*])));
  for *f* ← *Low*(*FormulaSort*) **to** *High*(*FormulaSort*) **do**
    FormulaKindLookupTable.*Insert*(*new*(*MStrPtr*, *Init*(*FormulaName*[*f*])));
  for *t* ← *Low*(*TermSort*) **to** *High*(*TermSort*) **do**
    TermKindLookupTable.*Insert*(*new*(*MStrPtr*, *Init*(*TermName*[*t*])));
  for *i* ← *itDefPred* **to** *itDefStruct* **do**
    PatternKindLookupTable.*Insert*(*new*(*MStrPtr*, *Init*(*DefPatternName*[*i*])));
  for *p* ← *Low*(*PropertyKind*) **to** *High*(*PropertyKind*) **do**
    PropertyKindLookupTable.*Insert*(*new*(*MStrPtr*, *Init*(*PropertyName*[*p*])));
  for *c* ← *Low*(*CorrectnessKind*) **to** *High*(*CorrectnessKind*) **do**
    CorrectnessKindLookupTable.*Insert*(*new*(*MStrPtr*, *Init*(*CorrectnessName*[*c*])));
  end;

1102. We also need to free the memory consumed by the lookup tables.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *DisposeWSLookupTables*;
  begin *ElemLookupTable*.*Done*; *AttrLookupTable*.*Done*; *BlockLookupTable*.*Done*;
  ItemLookupTable.*Done*; *FormulaKindLookupTable*.*Done*; *TermKindLookupTable*.*Done*;
  CorrectnessKindLookupTable.*Done*; *PropertyKindLookupTable*.*Done*;
  end;

1103. We can recall, from the XML dictionary module (§462), the different kinds of XML elements as specified by an enumerated constant. This converts the "`nr`" attribute to the human readable equivalents.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *Str2XMLElemKind*(*aStr* : *string*): *XMLElemKind*;
  var *lNr*: *integer*;
  begin *lNr* ← *ElemLookupTable*.*IndexOfStr*(*aStr*);
  if *lNr* > −1 **then** *Str2XMLElemKind* ← *XMLElemKind*(*lNr*)
  else *Str2XMLElemKind* ← *elUnknown*;
  end;

1104. Like the previous function, this converts the "`nr`" attribute for a WSM Mizar attribute XML element into a human readable form.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *Str2XMLAttrKind*(*aStr* : *string*): *XMLAttrKind*;
  var *lNr*: *integer*;
  begin *lNr* ← *AttrLookupTable*.*IndexOfStr*(*aStr*);
  if *lNr* > −1 **then** *Str2XMLAttrKind* ← *XMLAttrKind*(*lNr*)
  else *Str2XMLAttrKind* ← *atUnknown*;
  end;

1105. The "kinds" of different syntactic classes were introduced earlier in `wsmarticle.pas`, now we want to translate them into human readable form.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *Str2BlockKind*(*aStr* : *string*): *BlockKind*;
  var *lNr*: *integer*;
  begin *lNr* ← *BlockLookupTable*.*IndexOfStr*(*aStr*);
  if *lNr* > −1 **then** *Str2BlockKind* ← *BlockKind*(*lNr*)
  else *Str2BlockKind* ← *blMain*;
  end;
function *Str2ItemKind*(*aStr* : *string*): *ItemKind*;
  var *lNr*: *integer*;
  begin *lNr* ← *ItemLookupTable*.*IndexOfStr*(*aStr*);
  if *lNr* > −1 **then** *Str2ItemKind* ← *ItemKind*(*lNr*)
  else *Str2ItemKind* ← *itIncorrItem*;
  end;
function *Str2PatterenKind*(*aStr* : *string*): *ItemKind*;
  var *lNr*: *integer*;
  begin *lNr* ← *PatternKindLookupTable*.*IndexOfStr*(*aStr*);
  if *lNr* > −1 **then** *Str2PatterenKind* ← *ItemKind*(*Ord*(*ItDefPred*) + *lNr*)
  else *Str2PatterenKind* ← *itIncorrItem*;
  end;
function *Str2FormulaKind*(*aStr* : *string*): *FormulaSort*;
  var *lNr*: *integer*;
  begin *lNr* ← *FormulaKindLookupTable*.*IndexOfStr*(*aStr*);
  if *lNr* > −1 **then** *Str2FormulaKind* ← *FormulaSort*(*lNr*)
  else *Str2FormulaKind* ← *wsErrorFormula*;
  end;

1106.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *Str2TermKind*(*aStr* : *string*): *TermSort*;
  var *lNr*: *integer*;
  begin *lNr* ← *TermKindLookupTable.IndexOfStr*(*aStr*);
  if *lNr* > −1 **then** *Str2TermKind* ← *TermSort*(*lNr*)
  else *Str2TermKind* ← *wsErrorTerm*;
  end;
function *Str2PropertyKind*(*aStr* : *string*): *PropertyKind*;
  var *lNr*: *integer*;
  begin *lNr* ← *PropertyKindLookupTable.IndexOfStr*(*aStr*);
  if *lNr* > −1 **then** *Str2PropertyKind* ← *PropertyKind*(*lNr*)
  end;
function *Str2CorrectnessKind*(*aStr* : *string*): *CorrectnessKind*;
  var *lNr*: *integer*;
  begin *lNr* ← *CorrectnessKindLookupTable.IndexOfStr*(*aStr*);
  if *lNr* > −1 **then** *Str2CorrectnessKind* ← *CorrectnessKind*(*lNr*)
  end;

Subsection 21.12.1. Parsing types

1107. Reading a "term list" just iteratively invokes *Read_Term* (§1121) until all the children have been read.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_TermList*: *PList*;
  begin *result* ← *new*(*PList*, *Init*(0));
  while *nState* ≠ *eEnd* **do** *result*↑.*Insert*(*Read_Term*);
  end;

1108. An adjective is either "positive" (i.e., not negated) or "negative" (i.e., negated). We handle the first case in the "true" branch, and the second case in the "false" branch.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_Adjective*: *AdjectiveExpressionPtr*;
  var *lAttrNr*: *integer*; *lPos*: *Position*; *lNoneOcc*: *Boolean*;
  begin if *nElName* = *AdjectiveSortName*[*wsAdjective*] **then**
    begin *lPos* ← *GetAttrPos*; *lAttrNr* ← *GetIntAttr*(*XMLAttrName*[*atNr*]); *NextElementState*;
    result ← *new*(*AdjectivePtr*, *Init*(*lPos*, *lAttrNr*, *Read_TermList*)); *NextElementState*;
    end
  else begin *lPos* ← *GetAttrPos*; *NextElementState*;
    result ← *new*(*NegatedAdjectivePtr*, *Init*(*lPos*, *Read_Adjective*)); *NextElementState*;
    end;
  end;

1109. Reading a list of adjectives just iterates over the children of an element.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_AdjectiveList*: *PList*;
  begin *result* ← *new*(*Plist*, *Init*(0)); *NextElementState*;
  while *nState* ≠ *eEnd* **do** *result*↑.*Insert*(*Read_Adjective*);
  NextElementState;
  end;

1110. There are three valid Mizar types: "standard" types, structure types, and expandable modes (i.e., a cluster of adjectives stacked atop a type). If the XML element fails to match these three, then we should produce an "incorrect type".

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

function *InWSMizFileObj.Read_Type*: *TypePtr*;
  var *lList*: *Plist*; *lPos*: *Position*; *lModeSymbol*: *integer*;
  begin if *nElName* = *TypeName*[*wsStandardType*] **then**
    begin *lPos* ← *GetAttrPos*; *lModeSymbol* ← *GetIntAttr*(*XMLAttrName*[*atNr*]); *NextElementState*;
    result ← *new*(*StandardTypePtr*, *Init*(*lPos*, *lModeSymbol*, *Read_TermList*)); *NextElementState*;
    end
  else if *nElName* = *TypeName*[*wsStructureType*] **then**
      begin *lPos* ← *GetAttrPos*; *lModeSymbol* ← *GetIntAttr*(*XMLAttrName*[*atNr*]); *NextElementState*;
      result ← *new*(*StructTypePtr*, *Init*(*lPos*, *lModeSymbol*, *Read_TermList*)); *NextElementState*;
      end
    else if *nElName* = *TypeName*[*wsClusteredType*] **then**
        begin *lPos* ← *GetAttrPos*; *NextElementState*; *lList* ← *Read_AdjectiveList*;
        result ← *new*(*ClusteredTypePtr*, *Init*(*lPos*, *lList*, *Read_Type*)); *NextElementState*;
        end
      else begin *lPos* ← *GetAttrPos*; *NextElementState*; *result* ← *new*(*IncorrectTypePtr*, *Init*(*lPos*));
        NextElementState;
        end
  end;

Subsection 21.12.2. Parsing formulas

1111. Parsing a variable from XML just requires reading the attributes, since it is an empty-element.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

function *InWSMizFileObj.Read_Variable*: *VariablePtr*;
  var *lPos*: *Position*; *lNr*: *integer*;
  begin *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atIdNr*]);
  NextElementState; { closes the variable's tag }
  result ← *new*(*VariablePtr*, *Init*(*lPos*, *lNr*));
  NextElementState; { starts the next tag }
  end;

1112. Implicitly qualified variables are just wrappers around a variable.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

function *InWSMizFileObj.Read_ImplicitlyQualifiedSegment*: *ImplicitlyQualifiedSegmentPtr*;
  var *lPos*: *Position*;
  begin *lPos* ← *GetAttrPos*; *NextElementState*;
  result ← *new*(*ImplicitlyQualifiedSegmentPtr*, *Init*(*lPos*, *Read_Variable*)); *NextElementState*;
  end;

1113. Recall (§1025) that a "qualified segment" is either implicit (i.e., a wrapper around a single variable) or explicit (i.e., an element whose children are variables and a type).

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_VariableSegment*: *QualifiedSegmentPtr*;
  var *lPos*: *Position*; *lVar*: *VariablePtr*; *lList*: *PList*;
  begin if *nElName* = *SegmentKindName*[*ikImplQualifiedSegm*] **then**
    begin *result* ← *Read_ImplicitlyQualifiedSegment*;
    end
  else if *nElName* = *SegmentKindName*[*ikExplQualifiedSegm*] **then**
      begin *lPos* ← *GetAttrPos*; *NextElementState*; *lList* ← *new*(*PList*, *Init*(0));
      NextElementState; { read the variables }
      while (*nState* = *eStart*) ∧ (*nElName* = *XMLElemName*[*elVariable*]) **do**
        lList↑.*Insert*(*Read_Variable*);
      NextElementState; { read the type }
      result ← *new*(*ExplicitlyQualifiedSegmentPtr*, *Init*(*lPos*, *lList*, *Read_Type*));
      NextElementState; { start the next tag }
      end
  end;

1114. Private predicates are empty elements, so we only need to read their attributes.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_PrivatePredicativeFormula*: *PrivatePredicativeFormulaPtr*;
  var *lPos*: *Position*; *lNr*: *integer*;
  begin *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atIdNr*]); *NextElementState*;
  Result ← *new*(*PrivatePredicativeFormulaPtr*, *Init*(*lPos*, *lNr*, *Read_TermList*)); *NextElementState*;
  end;

1115.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_Formula*: *FormulaPtr*;
  var *lPos*: *Position*; *lNr*: *integer*; *lList*: *PList*; *lFrm*: *FormulaPtr*; *lTrm*: *TermPtr*;
    lSgm: *QualifiedSegmentPtr*;
  begin case *Str2FormulaKind*(*nElName*) **of**
  wsNegatedFormula: **begin** *lPos* ← *GetAttrPos*; *NextElementState*;
    result ← *new*(*NegativeFormulaPtr*, *Init*(*lPos*, *Read_Formula*)); *NextElementState*;
    end;
  ⟨ Parse XML for formula with binary connective 1116 ⟩;
  wsFlexaryConjunctiveFormula: **begin** *lPos* ← *GetAttrPos*; *NextElementState*; *lFrm* ← *Read_Formula*;
    result ← *new*(*FlexaryConjunctiveFormulaPtr*, *Init*(*lPos*, *lFrm*, *Read_Formula*)); *NextElementState*;
    end;
  wsFlexaryDisjunctiveFormula: **begin** *lPos* ← *GetAttrPos*; *NextElementState*; *lFrm* ← *Read_Formula*;
    result ← *new*(*FlexaryDisjunctiveFormulaPtr*, *Init*(*lPos*, *lFrm*, *Read_Formula*)); *NextElementState*;
    end;
  ⟨ Parse XML for predicate-based formula 1117 ⟩;
  wsAttributiveFormula: **begin** *lPos* ← *GetAttrPos*; *NextElementState*; *lTrm* ← *Read_Term*;
    Result ← *new*(*AttributiveFormulaPtr*, *Init*(*lPos*, *lTrm*, *Read_AdjectiveList*)); *NextElementState*;
    end;
  wsQualifyingFormula: **begin** *lPos* ← *GetAttrPos*; *NextElementState*; *lTrm* ← *Read_Term*;
    Result ← *new*(*QualifyingFormulaPtr*, *Init*(*lPos*, *lTrm*, *Read_Type*)); *NextElementState*;
    end;
  wsUniversalFormula: **begin** *lPos* ← *GetAttrPos*; *NextElementState*; *lSgm* ← *Read_VariableSegment*;
    Result ← *new*(*UniversalFormulaPtr*, *Init*(*lPos*, *lSgm*, *Read_Formula*)); *NextElementState*;
    end;
  wsExistentialFormula: **begin** *lPos* ← *GetAttrPos*; *NextElementState*; *lSgm* ← *Read_VariableSegment*;
    Result ← *new*(*ExistentialFormulaPtr*, *Init*(*lPos*, *lSgm*, *Read_Formula*)); *NextElementState*;
    end;
  wsContradiction: **begin** *lPos* ← *GetAttrPos*; *NextElementState*;
    result ← *new*(*ContradictionFormulaPtr*, *Init*(*lPos*)); *NextElementState*;
    end;
  wsThesis: **begin** *lPos* ← *GetAttrPos*; *NextElementState*; *result* ← *new*(*ThesisFormulaPtr*, *Init*(*lPos*));
    NextElementState;
    end;
  wsErrorFormula: **begin** *lPos* ← *GetAttrPos*; *NextElementState*;
    result ← *new*(*IncorrectFormulaPtr*, *Init*(*lPos*)); *NextElementState*;
    end;
  endcases;
  end;

1116. For formulas with binary connectives, we read both arguments.

⟨ Parse XML for formula with binary connective 1116 ⟩ ≡

wsConjunctiveFormula: **begin** *lPos* ← *GetAttrPos*; *NextElementState*; *lFrm* ← *Read_Formula*;
result ← *new*(*ConjunctiveFormulaPtr*, *Init*(*lPos*, *lFrm*, *Read_Formula*)); *NextElementState*;
end;
wsDisjunctiveFormula: **begin** *lPos* ← *GetAttrPos*; *NextElementState*; *lFrm* ← *Read_Formula*;
result ← *new*(*DisjunctiveFormulaPtr*, *Init*(*lPos*, *lFrm*, *Read_Formula*)); *NextElementState*;
end;
wsConditionalFormula: **begin** *lPos* ← *GetAttrPos*; *NextElementState*; *lFrm* ← *Read_Formula*;
result ← *new*(*ConditionalFormulaPtr*, *Init*(*lPos*, *lFrm*, *Read_Formula*)); *NextElementState*;
end;
wsBiconditionalFormula: **begin** *lPos* ← *GetAttrPos*; *NextElementState*; *lFrm* ← *Read_Formula*;
result ← *new*(*BiconditionalFormulaPtr*, *Init*(*lPos*, *lFrm*, *Read_Formula*)); *NextElementState*;
end

This code is used in section 1115.

1117.

⟨ Parse XML for predicate-based formula 1117 ⟩ ≡

wsPredicativeFormula: **begin** *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atNr*]);
NextElementState; *NextElementState*; { Arguments }
lList ← *Read_TermList*; *NextElementState*; { Arguments }
NextElementState; { Arguments }
Result ← *new*(*PredicativeFormulaPtr*, *Init*(*lPos*, *lNr*, *lList*, *Read_TermList*)); *NextElementState*;
NextElementState;
end;
wsRightSideOfPredicativeFormula: **begin** *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atNr*]);
NextElementState; *NextElementState*; { Arguments }
Result ← *new*(*RightSideOfPredicativeFormulaPtr*, *Init*(*lPos*, *lNr*, *Read_TermList*)); *NextElementState*;
NextElementState;
end;
wsMultiPredicativeFormula: **begin** *lPos* ← *GetAttrPos*; *NextElementState*; *lList* ← *new*(*PList*, *Init*(0));
while *nState* ≠ *eEnd* **do** *lList*↑.*Insert*(*Read_Formula*);
result ← *new*(*MultiPredicativeFormulaPtr*, *Init*(*lPos*, *lList*)); *NextElementState*;
end;
wsPrivatePredicateFormula: **begin** *Result* ← *Read_PrivatePredicativeFormula*;
end

This code is used in section 1115.

Subsection 21.12.3. Parsing terms

1118.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_SimpleTerm*: *SimpleTermPtr*;
  var *lPos*: *Position*; *lNr*: *integer*;
  begin *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atIdNr*]); *NextElementState*;
  result ← *new*(*SimpleTermPtr*, *Init*(*lPos*, *lNr*)); *NextElementState*;
  end;

1119.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_PrivateFunctorTerm*: *PrivateFunctorTermPtr*;
  var *lPos*: *Position*; *lNr*: *integer*;
  begin *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atIdNr*]); *NextElementState*;
  result ← *new*(*PrivateFunctorTermPtr*, *Init*(*lPos*, *lNr*, *Read_TermList*)); *NextElementState*;
  end;

1120.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_InternalSelectorTerm*: *InternalSelectorTermPtr*;
  var *lPos*: *Position*; *lNr*: *integer*;
  begin *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atNr*]); *NextElementState*;
  result ← *new*(*InternalSelectorTermPtr*, *Init*(*lPos*, *lNr*)); *NextElementState*;
  end;

1121.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_Term*: *TermPtr*;
  var *lPos*, *lRPos*: *Position*; *lNr*, *lRNr*: *integer*; *lList*: *PList*; *lTrm*: *TermPtr*;
  begin case *Str2TermKind*(*nElName*) **of**
  wsPlaceholderTerm: **begin** *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atNr*]);
    NextElementState; *result* ← *new*(*PlaceholderTermPtr*, *Init*(*lPos*, *lNr*)); *NextElementState*;
    end;
  wsSimpleTerm: **begin** *result* ← *Read_SimpleTerm*;
    end;
  wsNumeralTerm: **begin** *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atNumber*]);
    NextElementState; *result* ← *new*(*NumeralTermPtr*, *Init*(*lPos*, *lNr*)); *NextElementState*;
    end;
  wsInfixTerm: **begin** *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atNr*]); *NextElementState*;
    NextElementState; { Arguments }
    lList ← *Read_TermList*; *NextElementState*; { Arguments }
    NextElementState; { Arguments }
    result ← *new*(*InfixTermPtr*, *Init*(*lPos*, *lNr*, *lList*, *Read_TermList*)); *NextElementState*;
    NextElementState;
    end;
  wsCircumfixTerm: **begin** *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atNr*]);
    NextElementState; *NextElementState*; *lRNr* ← *GetIntAttr*(*XMLAttrName*[*atNr*]);
    lRPos ← *GetAttrPos*; *NextElementState*;
    result ← *new*(*CircumfixTermPtr*, *Init*(*lPos*, *lNr*, *lRNr*, *Read_TermList*)); *NextElementState*;
    end;
  wsPrivateFunctorTerm: **begin** *result* ← *Read_PrivateFunctorTerm*;
    end;
  wsAggregateTerm: **begin** *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atNr*]);
    NextElementState; *result* ← *new*(*AggregateTermPtr*, *Init*(*lPos*, *lNr*, *Read_TermList*));
    NextElementState;
    end;
  wsSelectorTerm: **begin** *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atNr*]);
    NextElementState; *result* ← *new*(*SelectorTermPtr*, *Init*(*lPos*, *lNr*, *Read_Term*)); *NextElementState*;
    end;
  wsInternalSelectorTerm: *result* ← *Read_InternalSelectorTerm*;
  wsForgetfulFunctorTerm: **begin** *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atNr*]);
    NextElementState; *result* ← *new*(*ForgetfulFunctorTermPtr*, *Init*(*lPos*, *lNr*, *Read_Term*));
    NextElementState;
    end;
  wsInternalForgetfulFunctorTerm: **begin** *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atNr*]);
    NextElementState; *result* ← *new*(*InternalForgetfulFunctorTermPtr*, *Init*(*lPos*, *lNr*));
    NextElementState;
    end;
  wsFraenkelTerm: **begin** *lPos* ← *GetAttrPos*; *NextElementState*; *lList* ← *new*(*PList*, *Init*(0));
    while (*nState* = *eStart*) ∧ ((*nElName* = *SegmentKindName*[*ikImplQualifiedSegm*]) ∨ (*nElName* =
        SegmentKindName[*ikExplQualifiedSegm*])) **do** *lList*↑.*Insert*(*Read_VariableSegment*);
    lTrm ← *Read_Term*; *result* ← *new*(*FraenkelTermPtr*, *Init*(*lPos*, *lList*, *lTrm*, *Read_Formula*));
    NextElementState;
    end;
  wsSimpleFraenkelTerm: **begin** *lPos* ← *GetAttrPos*; *NextElementState*; *lList* ← *new*(*PList*, *Init*(0));
    while (*nState* = *eStart*) ∧ ((*nElName* = *SegmentKindName*[*ikImplQualifiedSegm*]) ∨ (*nElName* =
        SegmentKindName[*ikExplQualifiedSegm*])) **do** *lList*↑.*Insert*(*Read_VariableSegment*);

$lTrm \leftarrow Read\_Term$; $result \leftarrow new(SimpleFraenkelTermPtr, Init(lPos, lList, lTrm))$;
$NextElementState$;
end;
$wsQualificationTerm$: **begin** $lPos \leftarrow GetAttrPos$; $NextElementState$; $lTrm \leftarrow Read\_Term$;
$Result \leftarrow new(QualifiedTermPtr, Init(lPos, lTrm, Read\_Type))$; $NextElementState$;
end;
$wsExactlyTerm$: **begin** $lPos \leftarrow GetAttrPos$; $NextElementState$;
$Result \leftarrow new(ExactlyTermPtr, Init(lPos, Read\_Term))$; $NextElementState$;
end;
$wsGlobalChoiceTerm$: **begin** $lPos \leftarrow GetAttrPos$; $NextElementState$;
$Result \leftarrow new(ChoiceTermPtr, Init(lPos, Read\_Type))$; $NextElementState$;
end;
$wsItTerm$: **begin** $lPos \leftarrow GetAttrPos$; $NextElementState$; $Result \leftarrow new(ItTermPtr, Init(lPos))$;
$NextElementState$;
end;
$wsErrorTerm$: **begin** $lPos \leftarrow GetAttrPos$; $NextElementState$;
$Result \leftarrow new(IncorrectTermPtr, Init(lPos))$; $NextElementState$;
end;
endcases;
end;

Subsection 21.12.4. Parsing text items

1122.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function $InWSMizFileObj.Read\_TypeList$: $PList$;
begin $NextElementState$; $result \leftarrow new(PList, Init(0))$;
while $nState \neq eEnd$ **do** $result\uparrow.Insert(Read\_Type)$;
$NextElementState$;
end;

1123.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function $InWSMizFileObj.Read\_Locus$: $LocusPtr$;
var $lPos$: $Position$; lNr: $integer$;
begin $lPos \leftarrow GetAttrPos$; $lNr \leftarrow GetIntAttr(XMLAttrName[atIdNr])$; $NextElementState$;
$result \leftarrow new(LocusPtr, Init(lPos, lNr))$; $NextElementState$;
end;

1124.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function $InWSMizFileObj.Read\_Loci$: $PList$;
begin $NextElementState$; $result \leftarrow new(PList, Init(0))$;
while $nState \neq eEnd$ **do** $result\uparrow.Insert(Read\_Locus)$;
$NextElementState$;
end;

1125.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_ModePattern*: *ModePatternPtr*;
  var *lPos*: *Position*; *lNr*: *integer*;
  begin *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atNr*]); *NextElementState*;
  result ← *new*(*ModePatternPtr*, *Init*(*lPos*, *lNr*, *Read_Loci*)); *NextElementState*;
  end;

1126.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_AttributePattern*: *AttributePatternPtr*;
  var *lPos*: *Position*; *lNr*: *integer*; *lArg*: *LocusPtr*;
  begin *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atNr*]); *NextElementState*;
  lArg ← *Read_Locus*; *result* ← *new*(*AttributePatternPtr*, *Init*(*lPos*, *lArg*, *lNr*, *Read_Loci*));
  NextElementState;
  end;

1127.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_FunctorPattern*: *FunctorPatternPtr*;
  var *lPos*, *lRPos*: *Position*; *lNr*, *lRNr*: *integer*; *lArgs*: *PList*;
  begin if *nState* = *eStart* **then**
    if *nElName* = *FunctorPatternName*[*InfixFunctor*] **then**
      begin *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atNr*]); *NextElementState*;
      lArgs ← *Read_Loci*; *result* ← *new*(*InfixFunctorPatternPtr*, *Init*(*lPos*, *lArgs*, *lNr*, *Read_Loci*));
      NextElementState;
      end
    else if *nElName* = *FunctorPatternName*[*CircumfixFunctor*] **then**
        begin *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atNr*]); *NextElementState*;
        lRNr ← *GetIntAttr*(*XMLAttrName*[*atNr*]); *NextElementState*; *NextElementState*;
        result ← *new*(*CircumfixFunctorPatternPtr*, *Init*(*lPos*, *lNr*, *lRNr*, *Read_Loci*)); *NextElementState*;
        end;
  end;

1128.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_PredicatePattern*: *PredicatePatternPtr*;
  var *lPos*, *lRPos*: *Position*; *lNr*, *lRNr*: *integer*; *lArgs*: *PList*;
  begin *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atNr*]); *NextElementState*;
  lArgs ← *Read_Loci*; *result* ← *new*(*PredicatePatternPtr*, *Init*(*lPos*, *lArgs*, *lNr*, *Read_Loci*));
  NextElementState;
  end;

1129.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_Pattern*: *PatternPtr*;
  begin case *Str2PatterenKind*(*nElName*) **of**
  itDefPred: *result* ← *Read_PredicatePattern*;
  itDefFunc: *result* ← *Read_FunctorPattern*;
  itDefMode: *result* ← *Read_ModePattern*;
  itDefAttr: *result* ← *Read_AttributePattern*;
  othercases if (*nElName* = *FunctorPatternName*[*InfixFunctor*]) ∨ (*nElName* = *FunctorPatternName*[*CircumfixFunctor*]) **then** *result* ← *Read_FunctorPattern*
    else *result* ← **nil**;
  endcases;
  end;

1130.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_Definiens*: *DefiniensPtr*;
  var *lPos*: *Position*; *lKind*, *lShape*: *string*; *lLab*: *LabelPtr*; *lExpr*: *PObject*; *lExpKind*: *ExpKind*;
    lList: *PList*; *lOtherwise*: *DefExpressionPtr*;
  begin *result* ← **nil**;
  if $(nState = eStart) \land (nElName = XMLElemName[elDefiniens])$ **then**
    begin *lPos* ← *GetAttrPos*; *lKind* ← *GetAttr*(*XMLAttrName*[*atKind*]);
    lShape ← *GetAttr*(*XMLAttrName*[*atShape*]); *NextElementState*; *lLab* ← *Read_Label*;
    if *lKind* = *DefiniensKindName*[*SimpleDefiniens*] **then**
      begin *lExpKind* ← *exFormula*;
      if *lShape* = *ExpName*[*exTerm*] **then** *lExpKind* ← *exTerm*;
      case *lExpKind* **of**
      exTerm: *lExpr* ← *Read_Term*;
      exFormula: *lExpr* ← *Read_Formula*;
      endcases;
      result ← *new*(*SimpleDefiniensPtr*, *Init*(*lPos*, *lLab*, *new*(*DefExpressionPtr*, *Init*(*lExpKind*, *lExpr*))));
      end
    else begin *lList* ← *new*(*Plist*, *Init*(0));
      while $(nState = eStart) \land (nElName = XMLElemName[elPartialDefiniens])$ **do**
        begin *NextElementState*; *lExpKind* ← *exFormula*;
        if *lShape* = *ExpName*[*exTerm*] **then** *lExpKind* ← *exTerm*;
        case *lExpKind* **of**
        exTerm: *lExpr* ← *Read_Term*;
        exFormula: *lExpr* ← *Read_Formula*;
        endcases; *lList*↑.*Insert*(*new*(*PartDefPtr*, *Init*(*new*(*DefExpressionPtr*, *Init*(*lExpKind*, *lExpr*)),
            Read_Formula))); *NextElementState*;
        end;
      lOtherwise ← **nil**;
      if $nState \neq eEnd$ **then**
        begin *lExpKind* ← *exFormula*;
        if *lShape* = *ExpName*[*exTerm*] **then** *lExpKind* ← *exTerm*;
        case *lExpKind* **of**
        exTerm: *lExpr* ← *Read_Term*;
        exFormula: *lExpr* ← *Read_Formula*;
        endcases; *lOtherwise* ← *new*(*DefExpressionPtr*, *Init*(*lExpKind*, *lExpr*));
        end;
      result ← *new*(*ConditionalDefiniensPtr*, *Init*(*lPos*, *lLab*, *lList*, *lOtherwise*))
      end;
    NextElementState;
    end;
  end;

1131.

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
function *InWSMizFileObj.Read_Label*: *LabelPtr*;
  var *lLabPos*: *Position*; *lLabId*: *Integer*;
  begin *result* ← **nil**;
  if $(nState = eStart) \land (nElName = XMLElemName[elLabel])$ **then**
    begin *lLabId* ← *GetIntAttr*(*XMLAttrName*[*atIdNr*]); *lLabPos* ← *GetAttrPos*; *NextElementState*;
    NextElementState; *result* ← *new*(*LabelPtr*, *Init*(*lLabId*, *lLabPos*));
    end;
  end;

1132.

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
function *InWSMizFileObj.Read_Proposition*: *PropositionPtr*;
  var *lPos*: *Position*; *lLab*: *LabelPtr*;
  begin *NextElementState*; *lLab* ← *Read_label*;
  result ← *new*(*PropositionPtr*, *Init*(*lLab*, *Read_Formula*, *lPos*)); *NextElementState*;
  end;

1133.

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
function *InWSMizFileObj.Read_LocalReference*: *LocalReferencePtr*;
  var *lPos*: *Position*; *lNr*: *integer*;
  begin *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atIdNr*]); *NextElementState*;
  NextElementState; *result* ← *new*(*LocalReferencePtr*, *Init*(*lNr*, *lPos*));
  end;

1134.

⟨Implementation for `wsmarticle.pas` 854⟩ +≡
function *InWSMizFileObj.Read_References*: *PList*;
  var *lPos*: *Position*; *lNr*, *lFileNr*: *integer*;
  begin *result* ← *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* ← *GetAttrPos*; *lFileNr* ← *GetIntAttr*(*XMLAttrName*[*atNr*]);
        lNr ← *GetIntAttr*(*XMLAttrName*[*atNumber*]); *NextElementState*; *NextElementState*;
        result↑.*Insert*(*new*(*TheoremReferencePtr*, *Init*(*lFileNr*, *lNr*, *lPos*)))
        end
      else if *nElName* = *ReferenceKindName*[*DefinitionReference*] **then**
          begin *lPos* ← *GetAttrPos*; *lFileNr* ← *GetIntAttr*(*XMLAttrName*[*atNr*]);
          lNr ← *GetIntAttr*(*XMLAttrName*[*atNumber*]); *NextElementState*; *NextElementState*;
          result↑.*Insert*(*new*(*DefinitionReferencePtr*, *Init*(*lFileNr*, *lNr*, *lPos*)))
          end;
  end;

1135.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_ReservationSegment*: *ReservationSegmentPtr*;
  var *lList*: *PList*;
  begin *lList* ← *new*(*PList*, *Init*(0)); *NextElementState*; { elVariables }
  while (*nState* = *eStart*) ∧ (*nElName* = *XMLElemName*[*elVariable*]) **do** *lList*↑.*Insert*(*Read_Variable*);
  NextElementState; *result* ← *new*(*ReservationSegmentPtr*, *Init*(*lList*, *Read_Type*));
  end;

1136.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_SchemeNameInSchemeHead*: *SchemePtr*;
  var *lNr*: *Integer*; *lPos*: *Position*;
  begin *lPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atIdNr*]);
  result ← *new*(*SchemePtr*, *Init*(*lNr*, *lPos*, **nil**, **nil**, **nil**));
  end;

1137.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_CompactStatement*: *CompactStatementPtr*;
  var *lProp*: *PropositionPtr*;
  begin *lProp* ← *Read_Proposition*; *result* ← *new*(*CompactStatementPtr*, *Init*(*lProp*, *Read_Justification*));
  end;

1138.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_StraightforwardJustification*: *StraightforwardJustificationPtr*;
  var *lPos*, *lLinkPos*: *Position*; *lLinked*: *boolean*;
  begin *lPos* ← *GetAttrPos*; *NextElementState*; *lLinked* ← *false*; *lLinkPos* ← *lPos*;
  if *nelName* = *XMLElemName*[*elLink*] **then**
    begin *lLinked* ← *true*; *lLinkPos* ← *GetAttrPos*; *NextElementState*; *NextElementState*;
    end;
  result ← *new*(*StraightforwardJustificationPtr*, *Init*(*lPos*, *lLinked*, *lLinkPos*));
  StraightforwardJustificationPtr(*result*)↑.*nReferences* ← *Read_References*; *NextElementState*;
  end;

1139.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_SchemeJustification*: *SchemeJustificationPtr*;
  var *lInfPos*, *lPos*: *Position*; *lNr*, *lIdNr*: *integer*;
  begin *lInfPos* ← *GetAttrPos*; *lNr* ← *GetIntAttr*(*XMLAttrName*[*atNr*]);
  lIdNr ← *GetIntAttr*(*XMLAttrName*[*atIdNr*]); *lPos.Line* ← *GetIntAttr*(*XMLAttrName*[*atPosLine*]);
  lPos.Col ← *GetIntAttr*(*XMLAttrName*[*atPosCol*]); *NextElementState*;
  result ← *new*(*SchemeJustificationPtr*, *Init*(*lInfPos*, *lNr*, *lIdNr*));
  SchemeJustificationPtr(*result*)↑.*nSchemeInfPos* ← *lPos*;
  SchemeJustificationPtr(*result*)↑.*nReferences* ← *Read_References*; *NextElementState*;
  end;

1140.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

function *InWSMizFileObj.Read_Justification*: *JustificationPtr*;
  var *lPos*: *Position*;
  begin if *nState* = *eStart* **then**
    if *nElName* = *InferenceName*[*infStraightforwardJustification*] **then**
      result ← *Read_StraightforwardJustification*
    else if *nElName* = *InferenceName*[*infSchemeJustification*] **then** *result* ← *Read_SchemeJustification*
      else if *nElName* = *InferenceName*[*infError*] **then**
          begin *lPos* ← *GetAttrPos*; *NextElementState*;
          result ← *new*(*JustificationPtr*, *Init*(*infError*, *lPos*)); *NextElementState*;
          end
        else if *nElName* = *InferenceName*[*infSkippedProof*] **then**
            begin *lPos* ← *GetAttrPos*; *NextElementState*;
            result ← *new*(*JustificationPtr*, *Init*(*infSkippedProof*, *lPos*)); *NextElementState*;
            end
          else *result* ← *new*(*JustificationPtr*, *Init*(*infProof*, *CurPos*));
  end;

1141.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

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* ← *Read_Label*; *result* ← *new*(*DiffuseStatementPtr*, *Init*(*lLab*, *stDiffuseStatement*));
    end
  else if *aShape* = *RegularStatementName*[*stCompactStatement*] **then**
      begin *result* ← *Read_CompactStatement*;
      end
    else if *aShape* = *RegularStatementName*[*stIterativeEquality*] **then**
        begin *lCStm* ← *Read_CompactStatement*; *result* ← *new*(*IterativeEqualityPtr*,
            Init(*lCStm*↑.*nProp*, *lCStm*↑.*nJustification*, *new*(*PList*, *Init*(0))));
        while (*nState* = *eStart*) ∧ (*nElName* = *XMLElemName*[*elIterativeStep*]) **do**
          begin *lPos* ← *GetAttrPos*; *NextElementState*; *lTrm* ← *Read_Term*;
          IterativeEqualityPtr(*result*)↑.*nIterSteps*↑.*Insert*(*new*(*IterativeStepPtr*, *Init*(*lPos*, *lTrm*,
              Read_Justification))); *NextElementState*;
          end;
        end;
  end;

1142.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *InWSMizFileObj.Read_ItemContentsAttr*(*aItem* : *wsItemPtr*; **var** *aShape* : *string*);
  begin *aShape* ← ´´;
  case *aItem*↑.*nItemKind* **of**
  itIncorrItem: ;
  itDefinition, *itSchemeBlock*, *itSchemeHead*, *itTheorem*, *itAxiom*, *itReservation*: ;
  itSection: ;
  itConclusion, *itRegularStatement*: *aShape* ← *GetAttr*(*XMLAttrName*[*atShape*]);
  itChoice, *itReconsider*, *itPrivFuncDefinition*, *itPrivPredDefinition*, *itConstantDefinition*, *itGeneralization*,
      itLociDeclaration, *itExistentialAssumption*, *itExemplification*, *itPerCases*, *itCaseBlock*: ;
  itCaseHead, *itSupposeHead*, *itAssumption*: ;
  itCorrCond: *aItem*↑.*nContent* ← *new*(*CorrectnessConditionPtr*, *Init*(*CurPos*,
      Str2CorrectnessKind(*GetAttr*(*XMLAttrName*[*atCondition*])), **nil**));
  itCorrectness: *aItem*↑.*nContent* ← *new*(*CorrectnessConditionsPtr*, *Init*(*CurPos*, [ ], **nil**));
  itProperty: *aShape* ← *GetAttr*(*XMLAttrName*[*atProperty*]);
  itDefFunc: *aShape* ← *GetAttr*(*XMLAttrName*[*atShape*]);
  itDefPred, *itDefMode*, *itDefAttr*, *itDefStruct*, *itPredSynonym*, *itPredAntonym*, *itFuncNotation*,
      itModeNotation, *itAttrSynonym*, *itAttrAntonym*, *itCluster*, *itIdentify*, *itReduction*: ;
  itPropertyRegistration: *aShape* ← *GetAttr*(*XMLAttrName*[*atProperty*]);
  itPragma: *aItem*↑.*nContent* ← *new*(*PragmaPtr*, *Init*(*XMLToStr*(*GetAttr*(*XMLAttrName*[*atSpelling*]))));
  endcases;
  end;

1143.

```
⟨ Implementation for wsmarticle.pas 854 ⟩ +≡
procedure InWSMizFileObj.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, lFieldSgmPos: Position;
    lRedefinition: boolean; lPattern: PatternPtr; lDef: HowToDefine; lPropertySort: PropertyKind;
  begin lPos ← CurPos;
  case aItem↑.nItemKind of
  itIncorrItem: ;
  itDefinition: ;
  itSchemeBlock: ;
  itSchemeHead: begin aItem↑.nContent ← Read_SchemeNameInSchemeHead; NextElementState;
    NextElementState; NextElementState;  { elSchematicVariables }
    lList ← new(PList, Init(0));
    while (nState = eStart) ∧ ((nElName = SchemeSegmentName[PredicateSegment]) ∨ (nElName =
          SchemeSegmentName[FunctorSegment])) do
      if nElName = SchemeSegmentName[PredicateSegment] then
        begin lPos ← GetAttrPos; NextElementState; lVars ← new(PList, Init(0)); NextElementState;
            { elVariables }
        while (nState = eStart) ∧ (nElName = XMLElemName[elVariable]) do
          lVars↑.Insert(Read_Variable);
        NextElementState;
        lList↑.Insert(new(PredicateSegmentPtr, Init(lPos, PredicateSegment, lVars, Read_TypeList)));
        NextElementState;
        end
      else begin lPos ← GetAttrPos; NextElementState; lVars ← new(PList, Init(0));
        NextElementState;  { elVariables }
        while (nState = eStart) ∧ (nElName = XMLElemName[elVariable]) do
          lVars↑.Insert(Read_Variable);
        NextElementState; lTyps ← Read_TypeList; NextElementState;
        lList↑.Insert(new(FunctorSegmentPtr, Init(lPos, lVars, lTyps, Read_Type))); NextElementState;
        NextElementState;
        end;
    SchemePtr(aItem↑.nContent)↑.nSchemeParams ← lList; NextElementState;
        { elSchematicVariables }
    SchemePtr(aItem↑.nContent)↑.nSchemeConclusion ← Read_Formula; lConds ← new(PList, Init(0));
    if (nState = eStart) ∧ (nElName = XMLElemName[elProvisionalFormulas]) then
      begin NextElementState;
      while (nState = eStart) ∧ (nElName = XMLElemName[elProposition]) do
        lConds↑.Insert(Read_Proposition);
      NextElementState;
      end;
    SchemePtr(aItem↑.nContent)↑.nSchemePremises ← lConds;
    end;
  itTheorem: aItem↑.nContent ← Read_CompactStatement;
  itAxiom: begin end;
  itReservation: aItem↑.nContent ← Read_ReservationSegment;
  itSection: ;
  itChoice: begin lList ← new(PList, Init(0));
    while (nState = eStart) ∧ ((nElName = SegmentKindName[ikImplQualifiedSegm]) ∨ (nElName =
          SegmentKindName[ikExplQualifiedSegm])) do lList↑.Insert(Read_VariableSegment);
```

```
    NextElementState; lConds ← nil;
    if nElName = XMLElemName[elProposition] then
      begin lConds ← new(PList, Init(0));
      while (nState = eStart) ∧ (nElName = XMLElemName[elProposition]) do
        lConds↑.Insert(Read_Proposition);
      end;
    NextElementState; aItem↑.nContent ← new(ChoiceStatementPtr, Init(lList, lConds,
        SimpleJustificationPtr(Read_Justification)));
    end;
  itReconsider: begin lList ← new(PList, Init(0));
    while (nState = eStart) ∧ ((nElName = XMLElemName[elEquality]) ∨ (nElName =
          XMLElemName[elVariable])) do
      if nElName = XMLElemName[elVariable] then
        lList↑.Insert(new(TypeChangePtr, Init(VariableIdentifier, Read_Variable, nil)))
      else begin NextElementState; lVar ← Read_Variable;
        lList↑.Insert(new(TypeChangePtr, Init(Equating, lVar, Read_Term))); NextElementState;
        end;
    lType ← Read_Type; aItem↑.nContent ← new(TypeChangingStatementPtr, Init(lList, lType,
        SimpleJustificationPtr(Read_Justification)));
    end;
  itPrivFuncDefinition: begin lVar ← Read_Variable; lList ← Read_TypeList;
    aItem↑.nContent ← new(PrivateFunctorDefinitionPtr, Init(lVar, lList, Read_Term));
    end;
  itPrivPredDefinition: begin lVar ← Read_Variable; lList ← Read_TypeList;
    aItem↑.nContent ← new(PrivatePredicateDefinitionPtr, Init(lVar, lList, Read_Formula));
    end;
  itConstantDefinition: begin lVar ← Read_Variable;
    aItem↑.nContent ← new(ConstantDefinitionPtr, Init(lVar, Read_Term));
    end;
  itLociDeclaration, itGeneralization: aItem↑.nContent ← Read_VariableSegment;
  itPerCases: aItem↑.nContent ← Read_Justification;
  itCaseBlock: ;
  itCorrCond: begin CorrectnessConditionPtr(aItem↑.nContent)↑.nJustification ← Read_Justification;
    end;
  itCorrectness: begin NextElementState;
    while (nState = eStart) ∧ (nElName = ItemName[itCorrectness]) do
      begin NextElementState; include(CorrectnessConditionsPtr(aItem↑.nContent)↑.nConditions,
          Str2CorrectnessKind(GetAttr(XMLAttrName[atCondition]))); NextElementState;
      end;
    NextElementState; CorrectnessConditionPtr(aItem↑.nContent)↑.nJustification ← Read_Justification;
    end;
  itProperty:
        aItem↑.nContent ← new(PropertyPtr, Init(lPos, Str2PropertyKind(aShape), Read_Justification));
  itConclusion, itRegularStatement: aItem↑.nContent ← Read_RegularStatement(aShape);
  itCaseHead, itSupposeHead, itAssumption: if nState = eStart then
      if nElName = AssumptionKindName[SingleAssumption] then
        begin lPos ← GetAttrPos; NextElementState;
        aItem↑.nContent ← new(SingleAssumptionPtr, Init(lPos, Read_Proposition)); NextElementState;
        end
      else if nElName = AssumptionKindName[CollectiveAssumption] then
          begin lPos ← GetAttrPos; NextElementState;
          aItem↑.nContent ← new(CollectiveAssumptionPtr, Init(lPos, new(PList, Init(0))));
```

```
        NextElementState;
        while (nState = eStart) ∧ (nElName = XMLElemName[elProposition]) do
          CollectiveAssumptionPtr(aItem↑.nContent)↑.nConditions↑.Insert(Read_Proposition);
        NextElementState; NextElementState;
        end;
itExistentialAssumption: begin aItem↑.nContent ← new(ExistentialAssumptionPtr, Init(lPos,
      new(PList, Init(0)), new(PList, Init(0))));
  while (nState = eStart) ∧ ((nElName = SegmentKindName[ikImplQualifiedSegm]) ∨ (nElName =
        SegmentKindName[ikExplQualifiedSegm])) do
    ExistentialAssumptionPtr(aItem↑.nContent)↑.nQVars↑.Insert(Read_VariableSegment);
  NextElementState;
  while (nState = eStart) ∧ (nElName = XMLElemName[elProposition]) do
    ExistentialAssumptionPtr(aItem↑.nContent)↑.nConditions↑.Insert(Read_Proposition);
  NextElementState;
  end;
itExemplification: begin lVar ← nil;
  if (nState = eStart) ∧ (nElName = XMLElemName[elVariable]) then lVar ← Read_Variable;
  lTrm ← nil;
  if nState ≠ eEnd then lTrm ← Read_Term;
  aItem↑.nContent ← new(ExamplePtr, Init(lVar, lTrm));
  end;
itDefPred: begin lRedefinition ← false;
  if (nState = eStart) ∧ (nElName = XMLElemName[elRedefine]) then
    begin NextElementState; NextElementState; lRedefinition ← true;
    end;
  lPattern ← Read_PredicatePattern; aItem↑.nContent ← new(PredicateDefinitionPtr, Init(lPos,
      lRedefinition, PredicatePatternPtr(lPattern), Read_Definiens));
  end;
itDefFunc: begin lRedefinition ← false;
  if (nState = eStart) ∧ (nElName = XMLElemName[elRedefine]) then
    begin NextElementState; NextElementState; lRedefinition ← true;
    end;
  lPattern ← Read_FunctorPattern; lType ← nil;
  if (nState = eStart) ∧ (nElName = XMLElemName[elTypeSpecification]) then
    begin NextElementState; lType ← Read_Type; NextElementState;
    end;
  if aShape = DefiningWayName[dfMeans] then lDef ← dfMeans
  else if aShape = DefiningWayName[dfEquals] then lDef ← dfEquals
    else lDef ← dfEmpty;
  case lDef of
  dfEquals: aItem↑.nContent ← new(FunctorDefinitionPtr, Init(lPos, lRedefinition,
        FunctorPatternPtr(lPattern), lType, lDef, Read_Definiens));
  dfMeans: aItem↑.nContent ← new(FunctorDefinitionPtr, Init(lPos, lRedefinition,
        FunctorPatternPtr(lPattern), lType, lDef, Read_Definiens));
  dfEmpty: aItem↑.nContent ← new(FunctorDefinitionPtr, Init(lPos, lRedefinition,
        FunctorPatternPtr(lPattern), lType, lDef, nil));
  endcases;
  end;
itDefMode: begin lRedefinition ← false;
  if (nState = eStart) ∧ (nElName = XMLElemName[elRedefine]) then
    begin NextElementState; NextElementState; lRedefinition ← true;
    end;
```

```
  lPattern ← Read_ModePattern;
  if (nState = eStart) ∧ (nElName = ModeDefinitionSortName[defExpandableMode]) then
    begin NextElementState; aItem↑.nContent ← new(ExpandableModeDefinitionPtr, Init(CurPos,
        ModePatternPtr(lPattern), Read_Type)); NextElementState;
    end
  else if (nState = eStart) ∧ (nElName = ModeDefinitionSortName[defStandardMode]) then
      begin NextElementState; lType ← nil;
      if (nState = eStart) ∧ (nElName = XMLElemName[elTypeSpecification]) then
        begin NextElementState; lType ← Read_Type; NextElementState;
        end;
      aItem↑.nContent ← new(StandardModeDefinitionPtr, Init(CurPos, lRedefinition,
          ModePatternPtr(lPattern), lType, Read_Definiens)); NextElementState;
      end;
  end;
itDefAttr: begin lRedefinition ← false;
  if (nState = eStart) ∧ (nElName = XMLElemName[elRedefine]) then
    begin NextElementState; NextElementState; lRedefinition ← true;
    end;
  lPattern ← Read_AttributePattern; aItem↑.nContent ← new(AttributeDefinitionPtr, Init(CurPos,
      lRedefinition, AttributePatternPtr(lPattern), Read_Definiens));
  end;
itDefStruct: begin NextElementState; lTyps ← new(PList, Init(0));
  while nState ≠ eEnd do lTyps↑.Insert(Read_Type);
  NextElementState; lPos ← GetAttrPos; lNr ← GetIntAttr(XMLAttrName[atNr]);
  NextElementState; lList ← nil;
  if (nState = eStart) ∧ (nElName = XMLElemName[elLoci]) then lList ← Read_Loci;
  lFields ← new(PList, Init(0));
  while (nState = eStart) ∧ (nElName = XMLElemName[elFieldSegment]) do
    begin lFieldSgmPos ← GetAttrPos; NextElementState; lSels ← new(PList, Init(0));
    while (nState = eStart) ∧ (nElName = XMLElemName[elSelector]) do
      begin lSels↑.Insert(new(FieldSymbolPtr, Init(GetAttrPos, GetIntAttr(XMLAttrName[atNr]))));
      NextElementState; NextElementState;
      end;
    lFields↑.Insert(new(FieldSegmentPtr, Init(lFieldSgmPos, lSels, Read_Type))); NextElementState;
    end;
  NextElementState;
  aItem↑.nContent ← new(StructureDefinitionPtr, Init(lPos, lTyps, lNr, lList, lFields));
  end;
itPredSynonym, itPredAntonym, itFuncNotation, itModeNotation, itAttrSynonym, itAttrAntonym: begin
      lPattern ← Read_Pattern; aItem↑.nContent ← new(NotationDeclarationPtr, Init(lPos,
      aItem↑.nItemKind, Read_Pattern, lPattern));
  end;
itCluster: if nState = eStart then
    if nElName = ClusterRegistrationName[ExistentialRegistration] then
      begin lPos ← GetAttrPos; NextElementState; lList ← Read_AdjectiveList;
      aItem↑.nContent ← new(EClusterPtr, Init(lPos, lList, Read_Type)); NextElementState;
      end
    else if nElName = ClusterRegistrationName[ConditionalRegistration] then
        begin lPos ← GetAttrPos; NextElementState; lList ← Read_AdjectiveList;
        lCons ← Read_AdjectiveList;
        aItem↑.nContent ← new(CClusterPtr, Init(lPos, lList, lCons, Read_Type)); NextElementState;
        end
```

```
        else if nElName = ClusterRegistrationName[FunctorialRegistration] then
            begin lPos ← GetAttrPos; NextElementState; lTrm ← Read_Term;
            lCons ← Read_AdjectiveList; lType ← nil;
            if nState ≠ eEnd then lType ← Read_Type;
            aItem↑.nContent ← new(FClusterPtr, Init(lPos, lTrm, lCons, lType)); NextElementState;
            end;
  itIdentify: begin lPattern ← Read_Pattern; aItem↑.nContent ← new(IdentifyRegistrationPtr,
        Init(lPos, Read_Pattern, lPattern, new(PList, Init(0))));
    while (nState = eStart) ∧ (nElName = XMLElemName[elLociEquality]) do
      begin lPos ← GetAttrPos; NextElementState; lLocus ← Read_Locus;
      IdentifyRegistrationPtr(aItem↑.nContent)↑.nEqLociList↑.Insert(new(LociEqualityPtr, Init(lPos,
          lLocus, Read_Locus))); NextElementState;
      end;
    end;
  itPropertyRegistration: begin lPropertySort ← Str2PropertyKind(aShape);
    case lPropertySort of
    sySethood: begin
          aItem↑.nContent ← new(SethoodRegistrationPtr, Init(lPos, lPropertySort, Read_Type));
      SethoodRegistrationPtr(aItem↑.nContent)↑.nJustification ← Read_Justification;
      end;
    endcases;
    end;
  itReduction: begin lTrm ← Read_Term;
    aItem↑.nContent ← new(ReduceRegistrationPtr, Init(lPos, Read_Term, lTrm));
    end;
  itPragma: ;
  endcases;
  end;
```

1144.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
function InWSMizFileObj.Read_TextProper: wsTextProperPtr;
  var lPos: Position;
  begin NextElementState; lPos.Line ← GetIntAttr(XMLAttrName[atLine]);
  lPos.Col ← GetIntAttr(XMLAttrName[atCol]); result ← new(wsTextProperPtr,
      Init(GetAttr(XMLAttrName[atArticleID]), GetAttr(XMLAttrName[atArticleExt]), lPos));
  if nDisplayInformationOnScreen then DisplayLine(result↑.nBlockPos.Line, 0);
  CurPos ← result↑.nBlockPos;
  if (nState = eStart) ∧ (nElName = BlockName[blMain]) then
    begin NextElementState;
    while (nState = eStart) ∧ (nElName = XMLElemName[elItem]) do
      result↑.nItems↑.Insert(Read_Item);
    end;
  NextElementState;
  end;
```

1145.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_Block*: *wsBlockPtr*;
  var *lPos*: *Position*;
  begin $lPos.Line \leftarrow GetIntAttr(XMLAttrName[atLine])$;
  $lPos.Col \leftarrow GetIntAttr(XMLAttrName[atCol])$;
  $result \leftarrow new(WSBlockPtr, Init(Str2BlockKind(GetAttr(XMLAttrName[atKind])), lPos))$;
  if *nDisplayInformationOnScreen* **then** $DisplayLine(result\uparrow.nBlockPos.Line, 0)$;
  $lPos.Line \leftarrow GetIntAttr(XMLAttrName[atPosLine])$;
  $lPos.Col \leftarrow GetIntAttr(XMLAttrName[atPosCol])$; $result\uparrow.nBlockEndPos \leftarrow lPos$;
  $CurPos \leftarrow result\uparrow.nBlockPos$; *NextElementState*;
  while $(nState = eStart) \land (nElName = XMLElemName[elItem])$ **do** $result\uparrow.nItems\uparrow.Insert(Read\_Item)$;
  $CurPos \leftarrow result\uparrow.nBlockEndPos$; *NextElementState*;
  end;

1146.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *InWSMizFileObj.Read_Item*: *wsItemPtr*;
  var *lStartTagNbr*: *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 $((nState = eEnd) \land (lStartTagNbr = 0)) \lor ((nState = eStart) \land (nElName = XMLElemName[elBlock]))$;
        if $(nState = eStart) \land (nElName = XMLElemName[elBlock])$ **then** $result\uparrow.nBlock \leftarrow Read\_Block$;
        end;
    end;
  $CurPos \leftarrow lPos$; *NextElementState*;
  end;

1147.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
function *Read_WSMizArticle*(*aFileName* : *string*): *wsTextProperPtr*;
  var *lInFile*: *InWSMizFilePtr*;
  begin *InitWSLookupTables*; $lInFile \leftarrow new(InWSMizFilePtr, OpenFile(aFileName))$;
  $result \leftarrow lInFile\uparrow.Read\_TextProper$; $dispose(lInFile, Done)$; *DisposeWSLookupTables*;
  end;

Section 21.13. PRETTYPRINTING WSM FILES (DEFERRED)

1148.

⟨ Publicly declared types in `wsmarticle.pas` 852 ⟩ +≡
  WSMizarPrinterPtr = ↑*WSMizarPrinterObj*;
  WSMizarPrinterObj = **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*;
    procedure *Print_AdjectiveList*(*aCluster* : *PList*); *virtual*;
    procedure *Print_Variable*(*aVar* : *VariablePtr*); *virtual*;
    procedure *Print_ImplicitlyQualifiedVariable*(*aSegm* : *ImplicitlyQualifiedSegmentPtr*); *virtual*;
    procedure *Print_VariableSegment*(*aSegm* : *QualifiedSegmentPtr*); *virtual*;
    procedure *Print_Type*(*aTyp* : *TypePtr*); *virtual*;
    procedure *Print_BinaryFormula*(*aFrm* : *BinaryFormulaPtr*); *virtual*;
    procedure *Print_PrivatePredicativeFormula*(*aFrm* : *PrivatePredicativeFormulaPtr*); *virtual*;
    procedure *Print_Formula*(*aFrm* : *FormulaPtr*); *virtual*;
    procedure *Print_OpenTermList*(*aTrmList* : *PList*); *virtual*;
    procedure *Print_TermList*(*aTrmList* : *PList*); *virtual*;
    procedure *Print_SimpleTermTerm*(*aTrm* : *SimpleTermPtr*); *virtual*;
    procedure *Print_PrivateFunctorTerm*(*aTrm* : *PrivateFunctorTermPtr*); *virtual*;
    procedure *Print_Term*(*aTrm* : *TermPtr*); *virtual*;
    procedure *Print_TypeList*(*aTypeList* : *PList*); *virtual*;
    procedure *Print_Label*(*aLab* : *LabelPtr*); *virtual*;
    procedure *Print_Reference*(*aRef* : *LocalReferencePtr*); *virtual*;
    procedure *Print_References*(*aRefs* : *PList*); *virtual*;
    procedure *Print_StraightforwardJustification*(*aInf* : *StraightforwardJustificationPtr*); *virtual*;
    procedure *Print_SchemeNameInJustification*(*aInf* : *SchemeJustificationPtr*); *virtual*;
    procedure *Print_SchemeJustification*(*aInf* : *SchemeJustificationPtr*); *virtual*;
    procedure *Print_Justification*(*aInf* : *JustificationPtr*; *aBlock* : *wsBlockPtr*); *virtual*;
    procedure *Print_Linkage*; *virtual*;
    procedure *Print_RegularStatement*(*aRStm* : *RegularStatementPtr*; *aBlock* : *wsBlockPtr*); *virtual*;
    procedure *Print_CompactStatement*(*aCStm* : *CompactStatementPtr*; *aBlock* : *wsBlockPtr*); *virtual*;
    procedure *Print_Proposition*(*aProp* : *PropositionPtr*); *virtual*;
    procedure *Print_Conditions*(*aCond* : *PList*);
    procedure *Print_AssumptionConditions*(*aCond* : *AssumptionPtr*); *virtual*;
    procedure *Print_Pattern*(*aPattern* : *PatternPtr*); *virtual*;
    procedure *Print_Locus*(*aLocus* : *LocusPtr*); *virtual*;
    procedure *Print_Loci*(*aLoci* : *PList*); *virtual*;
    procedure *Print_Definiens*(*aDef* : *DefiniensPtr*); *virtual*;
    procedure *Print_ReservedType*(*aResType* : *TypePtr*); *virtual*;
  end ;

1149. Constructor.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
constructor *WSMizarPrinterObj.OpenFile*(**const** *aFileName*: *string*);
  begin *inherited InitFile*(*AFileName*); *rewrite*(*nFile*); *nIndent* ← 0;
  nDisplayInformationOnScreen ← *false*;
  end;
destructor *WSMizarPrinterObj.Done*;
  begin *close*(*nFile*); *inherited Done*;
  end;

1150. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *WSMizarPrinterObj.Print_Char*(*aChar* : *char*);
  begin *write*(*nFile*, *aChar*);
  end;

1151. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *WSMizarPrinterObj.Print_NewLine*;
  begin *writeln*(*nFile*);
  end;

1152. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *WSMizarPrinterObj.Print_Number*(**const** *aNumber*: *integer*);
  begin *write*(*nFile*, *aNumber*); *Print_Char*(´␣´);
  end;

1153. The comment is translated from the Polish comment "?? czy na pewno trzeba robic konwersje", so I may be mistranslating.

⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *WSMizarPrinterObj.Print_String*(**const** *aString*: *string*);
  var *i*: *integer*;
  begin *write*(*nFile*, *XMLToStr*(*aString*)); { Do you really need to do conversions? }
  Print_Char(´␣´);
  end;

1154. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *WSMizarPrinterObj.Print_Indent*;
  var *i*: *integer*;
  begin for *i* ← 1 **to** *nIndent* **do** *Print_Char*(´␣´);
  end;

1155. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *WSMizarPrinterObj.Print_Adjective*(*aAttr* : *AdjectiveExpressionPtr*);
  begin case *aAttr*↑.*nAdjectiveSort* **of**
  wsAdjective: **with** *AdjectivePtr*(*aAttr*)↑ **do**
      begin if *nArgs*↑.*Count* ≠ 0 **then** *Print_TermList*(*nArgs*);
      Print_String(*AttributeName*[*nAdjectiveSymbol*]);
      end;
  wsNegatedAdjective: **begin** *Print_String*(*TokenName*[*sy_Non*]);
    Print_Adjective(*NegatedAdjectivePtr*(*aAttr*)↑.*nArg*);
    end;
  endcases;
  end;

1156. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
procedure WSMizarPrinterObj.Print_AdjectiveList(aCluster : PList);
  var i: integer;
  begin with aCluster↑ do
    for i ← 0 to Count − 1 do
      begin Print_Adjective(Items↑[i]);
      end;
  end;
```

1157. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
procedure WSMizarPrinterObj.Print_Variable(aVar : VariablePtr);
  begin with aVar↑ do
    begin Print_String(IdentRepr(nIdent));
    end;
  end;
```

1158. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
procedure WSMizarPrinterObj.Print_ImplicitlyQualifiedVariable(aSegm : ImplicitlyQualifiedSegmentPtr);
  begin Print_Variable(aSegm↑.nIdentifier);
  end;
```

1159. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
procedure WSMizarPrinterObj.Print_VariableSegment(aSegm : QualifiedSegmentPtr);
  var i: integer;
  begin case aSegm↑.nSegmentSort of
  ikImplQualifiedSegm: Print_ImplicitlyQualifiedVariable(ImplicitlyQualifiedSegmentPtr(aSegm));
  ikExplQualifiedSegm: with ExplicitlyQualifiedSegmentPtr(aSegm)↑ do
      begin Print_Variable(nIdentifiers.Items↑[0]);
      for i ← 1 to nIdentifiers↑.Count − 1 do
        begin Print_String(´,´); Print_Variable(nIdentifiers↑.Items↑[i]);
        end;
      Print_String(TokenName[sy_Be]); Print_Type(nType);
      end;
  endcases;
  end;
```

1160. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
procedure WSMizarPrinterObj.Print_OpenTermList(aTrmList : PList);
  var i: integer;
  begin if aTrmList↑.Count > 0 then
    begin Print_Term(aTrmList↑.Items↑[0]);
    for i ← 1 to aTrmList↑.Count − 1 do
      begin Print_String(´,´); Print_Term(aTrmList↑.Items↑[i]);
      end;
    end;
  end;
```

1161. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
procedure WSMizarPrinterObj.Print_TermList(aTrmList : PList);
  var i: integer;
  begin if aTrmList↑.Count > 0 then
    begin Print_String(´(´); Print_Term(aTrmList↑.Items↑[0]);
    for i ← 1 to aTrmList↑.Count − 1 do
      begin Print_String(´,´); Print_Term(aTrmList↑.Items↑[i]);
      end;
    Print_String(´)´);
    end;
  end;
```

1162. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
procedure WSMizarPrinterObj.Print_Type(aTyp : TypePtr);
  begin with aTyp↑ do
    begin case aTyp↑.nTypeSort of
    wsStandardType: with StandardTypePtr(aTyp)↑ do
        begin if nArgs↑.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)↑ do
        begin if nArgs↑.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)↑ do
        begin Print_AdjectiveList(nAdjectiveCluster); Print_Type(nType);
        end;
    wsErrorType: begin end;
    endcases;
    end;
  end;
```

1163. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
procedure WSMizarPrinterObj.Print_BinaryFormula(aFrm : BinaryFormulaPtr);
  begin Print_String(´(´); Print_Formula(aFrm↑.nLeftArg);
  case aFrm↑.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]);
    end;
  endcases; Print_Formula(aFrm↑.nRightArg); Print_String(´)´);
  end;
```

1164. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

procedure *WSMizarPrinterObj.Print_PrivatePredicativeFormula*(*aFrm* : *PrivatePredicativeFormulaPtr*);
  begin with *PrivatePredicativeFormulaPtr*(*aFrm*)↑ **do**
    begin *Print_String*(*IdentRepr*(*nPredIdNr*)); *Print_String*(´[´); *Print_OpenTermList*(*nArgs*);
    Print_String(´]´);
    end;
  end;

1165. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
procedure WSMizarPrinterObj.Print_Formula(aFrm : FormulaPtr);
  var i: Integer; lNeg: boolean; lFrm: FormulaPtr;
  begin case aFrm↑.nFormulaSort of
  wsNegatedFormula: begin Print_String(TokenName[sy_Not]);
    Print_Formula(NegativeFormulaPtr(aFrm)↑.nArg);
    end;
  wsConjunctiveFormula, wsDisjunctiveFormula, wsConditionalFormula,
        wsBiconditionalFormula, wsFlexaryConjunctiveFormula, wsFlexaryDisjunctiveFormula:
        Print_BinaryFormula(BinaryFormulaPtr(aFrm));
  wsPredicativeFormula: with PredicativeFormulaPtr(aFrm)↑ do
      begin Print_String('(');
      if nLeftArgs↑.Count ≠ 0 then
        begin Print_OpenTermList(nLeftArgs);
        end;
      Print_String(PredicateName[nPredNr]);
      if nRightArgs↑.Count ≠ 0 then
        begin Print_OpenTermList(nRightArgs);
        end;
      Print_String(')');
      end;
  wsMultiPredicativeFormula: with MultiPredicativeFormulaPtr(aFrm)↑ do
      begin Print_String('('); lFrm ← nScraps.Items↑[0];
      lNeg ← lFrm↑.nFormulaSort = wsNegatedFormula;
      if lNeg then lFrm ← NegativeFormulaPtr(lFrm)↑.nArg;
      with PredicativeFormulaPtr(lFrm)↑ do
        begin if nLeftArgs↑.Count ≠ 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↑.Count ≠ 0 then Print_OpenTermList(nRightArgs);
        end;
      for i ← 1 to nScraps.Count − 1 do
        begin lFrm ← nScraps.Items↑[i]; lNeg ← lFrm↑.nFormulaSort = wsNegatedFormula;
        if lNeg then lFrm ← NegativeFormulaPtr(lFrm)↑.nArg;
        with RightSideOfPredicativeFormulaPtr(lFrm)↑ do
          begin if lNeg then
            begin Print_String(TokenName[sy_Does]); Print_String(TokenName[sy_Not]);
            end;
          Print_String(PredicateName[nPredNr]);
          if nRightArgs↑.Count ≠ 0 then Print_OpenTermList(nRightArgs);
          end;
        end;
      Print_String(')');
      end;
  wsPrivatePredicateFormula: Print_PrivatePredicativeFormula(PrivatePredicativeFormulaPtr(aFrm));
  wsAttributiveFormula: with AttributiveFormulaPtr(aFrm)↑ do
      begin Print_String('('); Print_Term(nSubject); Print_String(TokenName[sy_Is]);
      Print_AdjectiveList(nAdjectives); Print_String(')');
      end;
  wsQualifyingFormula: with QualifyingFormulaPtr(aFrm)↑ do
```

```
      begin Print_String('('); Print_Term(nSubject); Print_String(TokenName[sy_Is]);
      Print_Type(nType); Print_String(')');
      end;
  wsUniversalFormula: with QuantifiedFormulaPtr(aFrm)↑ do
      begin Print_String('('); Print_String(TokenName[sy_For]);
      Print_VariableSegment(QuantifiedFormulaPtr(aFrm)↑.nSegment);
      Print_String(TokenName[sy_Holds]); Print_Formula(QuantifiedFormulaPtr(aFrm)↑.nScope);
      Print_String(')');
      end;
  wsExistentialFormula: with QuantifiedFormulaPtr(aFrm)↑ do
      begin Print_String('('); Print_String(TokenName[sy_Ex]);
      Print_VariableSegment(QuantifiedFormulaPtr(aFrm)↑.nSegment); Print_String(TokenName[sy_St]);
      Print_Formula(QuantifiedFormulaPtr(aFrm)↑.nScope); Print_String(')');
      end;
  wsContradiction: begin Print_String(TokenName[sy_Contradiction]);
    end;
  wsThesis: begin Print_String(TokenName[sy_Thesis]);
    end;
  wsErrorFormula: begin end;
  endcases;
  end;
```

1166. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
procedure WSMizarPrinterObj.Print_SimpleTermTerm(aTrm : SimpleTermPtr);
  begin Print_String(IdentRepr(SimpleTermPtr(aTrm)↑.nIdent));
  end;
```

1167. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
procedure WSMizarPrinterObj.Print_PrivateFunctorTerm(aTrm : PrivateFunctorTermPtr);
  begin Print_String(IdentRepr(aTrm↑.nFunctorIdent)); Print_String('(');
  Print_OpenTermList(aTrm↑.nArgs); Print_String(')');
  end;
```

1168. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

procedure *WSMizarPrinterObj.Print_Term*(*aTrm* : *TermPtr*);
  var *i*, *j*: *integer*; *lPrintWhere*: *boolean*;
  begin case *aTrm*↑.*nTermSort* **of**
  wsPlaceholderTerm: **begin** *Print_Char*(´$´); *Print_Number*(*PlaceholderTermPtr*(*aTrm*)↑.*nLocusNr*);
    end;
  wsSimpleTerm: **begin** *Print_SimpleTermTerm*(*SimpleTermPtr*(*aTrm*));
    end;
  wsNumeralTerm: **begin** *Print_Number*(*NumeralTermPtr*(*aTrm*)↑.*nValue*);
    end;
  wsInfixTerm: **with** *InfixTermPtr*(*aTrm*)↑ **do**
      begin *Print_String*(´(´);
      if *nLeftArgs*↑.*Count* ≠ 0 **then**
        begin *Print_TermList*(*nLeftArgs*);
        end;
      Print_String(*FunctorName*[*nFunctorSymbol*]);
      if *nRightArgs*↑.*Count* ≠ 0 **then**
        begin *Print_TermList*(*nRightArgs*);
        end;
      Print_String(´)´);
      end;
  wsCircumfixTerm: **with** *CircumfixTermPtr*(*aTrm*)↑ **do**
      begin *Print_String*(*LeftBracketName*[*nLeftBracketSymbol*]); *Print_OpenTermList*(*nArgs*);
      Print_String(*RightBracketName*[*nRightBracketSymbol*]);
      end;
  wsPrivateFunctorTerm: *Print_PrivateFunctorTerm*(*PrivateFunctorTermPtr*(*aTrm*));
  wsAggregateTerm: **with** *AggregateTermPtr*(*aTrm*)↑ **do**
      begin *Print_String*(*StructureName*[*nStructSymbol*]);
      Print_String(*TokenName*[*sy_StructLeftBracket*]); *Print_OpenTermList*(*nArgs*);
      Print_String(*TokenName*[*sy_StructRightBracket*]);
      end;
  wsSelectorTerm: **with** *SelectorTermPtr*(*aTrm*)↑ **do**
      begin *Print_String*(´(´); *Print_String*(*TokenName*[*sy_The*]);
      Print_String(*SelectorName*[*nSelectorSymbol*]); *Print_String*(*TokenName*[*sy_Of*]);
      Print_Term(*nArg*); *Print_String*(´)´);
      end;
  wsInternalSelectorTerm: **with** *InternalSelectorTermPtr*(*aTrm*)↑ **do**
      begin *Print_String*(*TokenName*[*sy_The*]); *Print_String*(*SelectorName*[*nSelectorSymbol*]);
      end;
  wsForgetfulFunctorTerm: **with** *ForgetfulFunctorTermPtr*(*aTrm*)↑ **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*)↑ **do**
      begin *Print_String*(´(´); *Print_String*(*TokenName*[*sy_The*]);
      Print_String(*StructureName*[*nStructSymbol*]); *Print_String*(´)´);
      end;
  wsFraenkelTerm: **with** *FraenkelTermPtr*(*aTrm*)↑ **do**
      begin *Print_String*(´{´); *Print_Term*(*nSample*);
      if *nPostqualification*↑.*Count* > 0 **then**
        begin *lPrintWhere* ← *true*;

```
        for i ← 0 to nPostqualification↑.Count − 1 do
          case QualifiedSegmentPtr(nPostqualification↑.Items↑[i])↑.nSegmentSort of
          ikImplQualifiedSegm: with ImplicitlyQualifiedSegmentPtr(nPostqualification↑.Items↑[i])↑ do
              begin Print_String(TokenName[sy_Where]); Print_Variable(nIdentifier);
              end;
          ikExplQualifiedSegm: with ExplicitlyQualifiedSegmentPtr(nPostqualification↑.Items↑[i])↑ do
              begin if lPrintWhere then
                begin Print_String(TokenName[sy_Where]); lPrintWhere ← false;
                end;
              Print_Variable(nIdentifiers.Items↑[0]);
              for j ← 1 to nIdentifiers↑.Count − 1 do
                begin Print_String(´,´); Print_Variable(nIdentifiers↑.Items↑[j]);
                end;
              Print_String(TokenName[sy_Is]); Print_Type(nType);
              if i < nPostqualification↑.Count − 1 then Print_String(´,´);
              end;
          endcases;
        end;
      Print_String(´:´); Print_Formula(nFormula); Print_String(´}´);
      end;
  wsSimpleFraenkelTerm: with SimpleFraenkelTermPtr(aTrm)↑ 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↑.Count > 0 then
        begin lPrintWhere ← true;
        for i ← 0 to nPostqualification↑.Count − 1 do
          case QualifiedSegmentPtr(nPostqualification↑.Items↑[i])↑.nSegmentSort of
          ikImplQualifiedSegm: with ImplicitlyQualifiedSegmentPtr(nPostqualification↑.Items↑[i])↑ do
              begin Print_String(TokenName[sy_Where]); Print_Variable(nIdentifier);
              end;
          ikExplQualifiedSegm: with ExplicitlyQualifiedSegmentPtr(nPostqualification↑.Items↑[i])↑ do
              begin if lPrintWhere then
                begin Print_String(TokenName[sy_Where]); lPrintWhere ← false;
                end;
              Print_Variable(nIdentifiers.Items↑[0]);
              for j ← 1 to nIdentifiers↑.Count − 1 do
                begin Print_String(´,´); Print_Variable(nIdentifiers↑.Items↑[j]);
                end;
              Print_String(TokenName[sy_Is]); Print_Type(nType);
              if i < nPostqualification↑.Count − 1 then Print_String(´,´);
              end;
          endcases;
        end;
      Print_String(´)´);
      end;
  wsQualificationTerm: with QualifiedTermPtr(aTrm)↑ do
      begin Print_String(´(´); Print_Term(nSubject); Print_String(TokenName[sy_Qua]);
      Print_Type(nQualification); Print_String(´)´);
      end;
  wsExactlyTerm: with ExactlyTermPtr(aTrm)↑ do
      begin Print_Term(nSubject); Print_String(TokenName[sy_Exactly]);
      end;
```

wsGlobalChoiceTerm: **begin** *Print_String*(´(´); *Print_String*(*TokenName*[*sy_The*]);
Print_Type(*ChoiceTermPtr*(*aTrm*)↑.*nChoiceType*); *Print_String*(´)´);
end;
wsItTerm: **begin** *Print_String*(*TokenName*[*sy_It*]);
end;
wsErrorTerm:
endcases;
end;

1169. ⟨Implementation for `wsmarticle.pas` 854⟩ +≡
procedure *WSMizarPrinterObj*.*Print_TypeList*(*aTypeList* : *PList*);
var *i*: *integer*;
begin if *aTypeList*↑.*Count* > 0 **then**
begin *Print_Type*(*aTypeList*↑.*Items*↑[0]);
for *i* ← 1 **to** *aTypeList*↑.*Count* − 1 **do**
begin *Print_String*(´,´); *Print_Type*(*aTypeList*↑.*Items*↑[*i*]);
end;
end;
end;

1170. ⟨Implementation for `wsmarticle.pas` 854⟩ +≡
procedure *WSMizarPrinterObj*.*Print_Label*(*aLab* : *LabelPtr*);
begin if (*aLab* ≠ **nil**) ∧ (*aLab*.*nLabelIdNr* > 0) **then**
begin *Print_String*(*IdentRepr*(*aLab*↑.*nLabelIdNr*)); *Print_String*(´:´);
end;
end;

1171. ⟨Implementation for `wsmarticle.pas` 854⟩ +≡
procedure *WSMizarPrinterObj*.*Print_Proposition*(*aProp* : *PropositionPtr*);
begin *Print_Label*(*aProp*↑.*nLab*); *Print_Formula*(*aProp*↑.*nSentence*);
end;

1172. ⟨Implementation for `wsmarticle.pas` 854⟩ +≡
procedure *WSMizarPrinterObj*.*Print_CompactStatement*(*aCStm* : *CompactStatementPtr*;
aBlock : *wsBlockPtr*);
begin with *aCStm*↑ **do**
begin *Print_Proposition*(*nProp*); *Print_Justification*(*nJustification*, *aBlock*);
end;
end;

1173. ⟨Implementation for `wsmarticle.pas` 854⟩ +≡
procedure *WSMizarPrinterObj*.*Print_Linkage*;
begin *Print_String*(*TokenName*[*sy_Then*]);
end;

1174. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *WSMizarPrinterObj.Print_RegularStatement*(*aRStm* : *RegularStatementPtr*;
    aBlock : *wsBlockPtr*);
  var *i*: *integer*;
  begin case *aRStm*↑.*nStatementSort* **of**
  stDiffuseStatement: **begin** *Print_Label*(*DiffuseStatementPtr*(*aRStm*)↑.*nLab*); *Print_Block*(*aBlock*);
    end;
  stCompactStatement: **begin**
      if (*CompactStatementPtr*(*aRStm*)↑.*nJustification*↑.*nInfSort* = *infStraightforwardJustification*) ∧
      StraightforwardJustificationPtr(*CompactStatementPtr*(*aRStm*)↑.*nJustification*)↑.*nLinked* **then**
      begin *Print_Linkage*;
      end;
    Print_CompactStatement(*CompactStatementPtr*(*aRStm*), *aBlock*);
    end;
  stIterativeEquality: **begin**
      if (*CompactStatementPtr*(*aRStm*)↑.*nJustification*↑.*nInfSort* = *infStraightforwardJustification*) ∧
      StraightforwardJustificationPtr(*CompactStatementPtr*(*aRStm*)↑.*nJustification*)↑.*nLinked* **then**
      begin *Print_Linkage*;
      end;
    Print_CompactStatement(*CompactStatementPtr*(*aRStm*), **nil**);
    with *IterativeEqualityPtr*(*aRStm*)↑ **do**
      for *i* ← 0 **to** *nIterSteps*↑.*Count* − 1 **do**
        with *IterativeStepPtr*(*nIterSteps*↑.*Items*↑[*i*])↑ **do**
          begin *Print_NewLine*; *Print_String*(*TokenName*[*sy_DotEquals*]); *Print_Term*(*nTerm*);
          Print_Justification(*nJustification*, **nil**);
          end;
    end;
  endcases;
  end;

1175. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *WSMizarPrinterObj.Print_Reference*(*aRef* : *LocalReferencePtr*);
  begin *Print_String*(*IdentRepr*(*aRef*↑.*nLabId*));
  end;

1176. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
procedure WSMizarPrinterObj.Print_References(aRefs : PList);
  var i: integer;
  begin for i ← 0 to aRefs↑.Count − 1 do
    with ReferencePtr(aRefs↑.Items↑[i])↑ do
      begin case nRefSort of
      LocalReference: begin Print_Reference(aRefs↑.Items↑[i]);
        end;
      TheoremReference: begin
            Print_String(MMLIdentifierName[TheoremReferencePtr(aRefs↑.Items↑[i])↑.nArticleNr]);
        Print_String(´:´); Print_Number(TheoremReferencePtr(aRefs↑.Items↑[i])↑.nTheoNr);
        end;
      DefinitionReference: begin
            Print_String(MMLIdentifierName[DefinitionReferencePtr(aRefs↑.Items↑[i])↑.nArticleNr]);
        Print_String(´:´); Print_String(´def´);
        Print_Number(DefinitionReferencePtr(aRefs↑.Items↑[i])↑.nDEfNr);
        end;
      endcases;
      if i < aRefs↑.Count − 1 then Print_String(´,´);
      end;
  end;
```

1177. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
procedure WSMizarPrinterObj.Print_StraightforwardJustification(aInf : StraightforwardJustificationPtr);
  begin with aInf↑ do
    begin if nReferences↑.Count ≠ 0 then
      begin Print_String(TokenName[sy_By]); Print_References(nReferences);
      end;
    end;
  end;
```

1178. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
procedure WSMizarPrinterObj.Print_SchemeNameInJustification(aInf : SchemeJustificationPtr);
  begin Print_String(IdentRepr(aInf↑.nSchemeIdNr));
  end;
```

1179. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
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↑.Count > 0 then
      begin Print_String(´(´); Print_References(nReferences); Print_String(´)´);
      end;
    end;
  end;
```

1180. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *WSMizarPrinterObj.Print_Justification*(*aInf* : *JustificationPtr*; *aBlock* : *wsBlockPtr*);
  begin case *aInf*↑.*nInfSort* **of**
  infStraightforwardJustification: *Print_StraightforwardJustification*(*StraightforwardJustificationPtr*(*aInf*));
  infSchemeJustification: *Print_SchemeJustification*(*SchemeJustificationPtr*(*aInf*));
  infError, *infSkippedProof*: **begin end**;
  infProof: *Print_Block*(*aBlock*);
  endcases;
  end;

1181. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *WSMizarPrinterObj.Print_Conditions*(*aCond* : *PList*);
  var *i*: *integer*;
  begin *Print_String*(*TokenName*[*sy_That*]); *Print_NewLine*; *Print_Proposition*(*aCond*↑.*Items*↑[0]);
  for $i \leftarrow 1$ **to** *aCond*↑.*Count* − 1 **do**
    begin *Print_String*(*TokenName*[*sy_And*]); *Print_NewLine*; *Print_Proposition*(*aCond*↑.*Items*↑[*i*]);
    end;
  end;

1182. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *WSMizarPrinterObj.Print_AssumptionConditions*(*aCond* : *AssumptionPtr*);
  begin case *aCond*↑.*nAssumptionSort* **of**
  SingleAssumption: **begin** *Print_Proposition*(*SingleAssumptionPtr*(*aCond*)↑.*nProp*);
    end;
  CollectiveAssumption: **begin** *Print_Conditions*(*CollectiveAssumptionPtr*(*aCond*)↑.*nConditions*);
    end;
  endcases;
  end;

1183. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *WSMizarPrinterObj.Print_Locus*(*aLocus* : *LocusPtr*);
  begin with *aLocus*↑ **do**
    begin *Print_String*(*IdentRepr*(*nVarId*));
    end;
  end;

1184. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *WSMizarPrinterObj.Print_Loci*(*aLoci* : *PList*);
  var *i*: *integer*;
  begin if (*aLoci* = **nil**) ∨ (*aLoci*↑.*Count* = 0) **then**
  else begin *Print_Locus*(*aLoci*↑.*Items*↑[0]);
    for $i \leftarrow 1$ **to** *aLoci*↑.*Count* − 1 **do**
      begin *Print_String*(´,´); *Print_Locus*(*aLoci*↑.*Items*↑[*i*]);
      end;
    end;
  end;

1185. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

procedure *WSMizarPrinterObj.Print_Pattern*(*aPattern* : *PatternPtr*);
  begin case *aPattern*↑.*nPatternSort* **of**
  itDefPred: **with** *PredicatePatternPtr*(*aPattern*)↑ **do**
      begin *Print_Loci*(*nLeftArgs*); *Print_String*(*PredicateName*[*nPredSymbol*]); *Print_Loci*(*nRightArgs*);
      end;
  itDefFunc: **begin case** *FunctorPatternPtr*(*aPattern*)↑.*nFunctKind* **of**
    InfixFunctor: **with** *InfixFunctorPatternPtr*(*aPattern*)↑ **do**
        begin if $(nLeftArgs \neq \mathbf{nil}) \land (nLeftArgs\uparrow.Count > 1)$ **then** *Print_String*(´(´);
        Print_Loci(*nLeftArgs*);
        if $(nLeftArgs \neq \mathbf{nil}) \land (nLeftArgs\uparrow.Count > 1)$ **then** *Print_String*(´)´);
        Print_String(*FunctorName*[*nOperSymb*]);
        if $(nRightArgs \neq \mathbf{nil}) \land (nRightArgs\uparrow.Count > 1)$ **then** *Print_String*(´(´);
        Print_Loci(*nRightArgs*);
        if $(nRightArgs \neq \mathbf{nil}) \land (nRightArgs\uparrow.Count > 1)$ **then** *Print_String*(´)´);
        end;
    CircumfixFunctor: **with** *CircumfixFunctorPatternPtr*(*aPattern*)↑ **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 $(nArgs \neq \mathbf{nil}) \land (nArgs\uparrow.Count > 0)$ **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;

1186. ⟨Implementation for `wsmarticle.pas` 854⟩ +≡

procedure *WSMizarPrinterObj.Print_Definiens*(*aDef* : *DefiniensPtr*);
  var *i*: *integer*;
  begin if *aDef* ≠ **nil then**
    with *DefiniensPtr*(*aDef*)↑ **do**
      begin case *nDefSort* **of**
      SimpleDefiniens: **begin if** (*nDefLabel* ≠ **nil**) ∧ (*nDefLabel*↑.*nLabelIdNr* > 0) **then**
          begin *Print_String*(´:´); *Print_Label*(*nDefLabel*);
          end;
        with *SimpleDefiniensPtr*(*aDef*)↑, *nExpression*↑ **do**
          case *nExprKind* **of**
          exTerm: *Print_Term*(*TermPtr*(*nExpr*));
          exFormula: *Print_Formula*(*FormulaPtr*(*nExpr*));
          endcases;
        end;
      ConditionalDefiniens: **begin if** (*nDefLabel* ≠ **nil**) ∧ (*nDefLabel*↑.*nLabelIdNr* > 0) **then**
          begin *Print_String*(´:´); *Print_Label*(*nDefLabel*);
          end;
        with *ConditionalDefiniensPtr*(*aDef*)↑ **do**
          begin for *i* ← 0 **to** *nConditionalDefiniensList*↑.*Count* − 1 **do**
            begin with *PartDefPtr*(*nConditionalDefiniensList*↑.*Items*↑[*I*])↑ **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* ≥ 0) ∧ (*i* < *nConditionalDefiniensList*↑.*Count* − 1) **then**
              begin *Print_String*(´,´); *Print_NewLine*;
              end;
            end;
          if *nOtherwise* ≠ **nil then**
            with *nOtherwise*↑ **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;

1187. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
procedure WSMizarPrinterObj.Print_Block(aWSBlock : WSBlockPtr);
  var i, lIndent: integer;
  begin with aWSBlock↑ do
    begin lIndent ← nIndent; Print_NewLine; Print_Indent;
    case nBlockKind of
    blDiffuse: begin Print_String(TokenName[sy_Now]); Print_NewLine;
      end;
    blHereby: begin Print_String(TokenName[sy_Now]); Print_NewLine;
      end;
    blProof: begin Print_String(TokenName[sy_Proof]); Print_NewLine;
      end;
    blDefinition: begin Print_String(TokenName[sy_Definition]); Print_NewLine;
      end;
    blNotation: begin Print_String(TokenName[sy_Notation]); Print_NewLine;
      end;
    blRegistration: begin Print_String(TokenName[sy_Registration]); Print_NewLine;
      end;
    blCase: Print_String(TokenName[sy_Case]);
    blSuppose: Print_String(TokenName[sy_Suppose]);
    blPublicScheme: ;
    endcases;
    for i ← 0 to nItems↑.Count − 1 do
      begin Print_Item(nItems↑.Items↑[i]);
      end;
    nIndent ← lIndent; Print_Indent; Print_String(TokenName[sy_End]);
    end;
  end;
```

1188. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
procedure WSMizarPrinterObj.Print_TextProper(aWSTextProper : WSTextProperPtr);
  var i: integer;
  begin with aWSTextProper↑ do
    begin for i ← 0 to nItems↑.Count − 1 do Print_Item(nItems↑.Items↑[i]);
    end;
  end;
```

1189. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
procedure WSMizarPrinterObj.Print_ReservedType(aResType : TypePtr);
  begin Print_Type(aResType);
  end;
```

1190. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

```
procedure WSMizarPrinterObj.Print_SchemeNameInSchemeHead(aSch : SchemePtr);
  begin Print_String(IdentRepr(aSch↑.nSchemeIdNr));
  end;
```

1191. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡

procedure *WSMizarPrinterObj.Print_Item*(*aWSItem* : *WSItemPtr*);
  var *i*, *j*, *lIndent*: *integer*;
  begin with *aWSItem*↑ **do**
    begin *CurPos* ← *nItemPos*;
    if *nDisplayInformationOnScreen* **then** *DisplayLine*(*CurPos.Line*, *ErrorNbr*);
    case *nItemKind* **of**
    itDefinition: **begin** *Print_Block*(*nBlock*); *Print_String*(´;´); *Print_NewLine*;
      end;
    itSchemeBlock: **begin** *Print_Block*(*nBlock*); *Print_String*(´;´); *Print_NewLine*;
      end;
    itSchemeHead: **with** *SchemePtr*(*nContent*)↑ **do**
        begin *Print_String*(*TokenName*[*sy_Scheme*]);
        Print_SchemeNameInSchemeHead(*SchemePtr*(*nContent*)); *Print_String*(´{´);
        for $j \leftarrow 0$ **to** *nSchemeParams*↑.*Count* − 1 **do**
          begin case *SchemeSegmentPtr*(*nSchemeParams*↑.*Items*↑[*j*])↑.*nSegmSort* **of**
          PredicateSegment: **with** *PredicateSegmentPtr*(*nSchemeParams*↑.*Items*↑[*j*])↑ **do**
              begin *Print_Variable*(*nVars*↑.*Items*↑[0]);
              for $i \leftarrow 1$ **to** *nVars*↑.*Count* − 1 **do**
                begin *Print_String*(´,´); *Print_Variable*(*nVars*↑.*Items*↑[*i*]);
                end;
              Print_String(´[´); *Print_TypeList*(*nTypeExpList*); *Print_String*(´]´);
              end;
          FunctorSegment: **with** *FunctorSegmentPtr*(*nSchemeParams*↑.*Items*↑[*j*])↑ **do**
              begin *Print_Variable*(*nVars*↑.*Items*↑[0]);
              for $i \leftarrow 1$ **to** *nVars.Count* − 1 **do**
                begin *Print_String*(´,´); *Print_Variable*(*nVars*↑.*Items*↑[*i*]);
                end;
              Print_String(´(´); *Print_TypeList*(*nTypeExpList*); *Print_String*(´)´);
              Print_String(*TokenName*[*sy_Arrow*]); *Print_Type*(*nSpecification*);
              end;
          endcases;
          if $(j \geq 0) \land (j <$ *nSchemeParams*↑.*Count* $- 1)$ **then** *Print_String*(´,´);
          end;
        Print_String(´}´); *Print_String*(´:´); *Print_Newline*; *Print_Formula*(*nSchemeConclusion*);
        Print_NewLine;
        if (*nSchemePremises* ≠ **nil**) ∧ (*nSchemePremises*↑.*Count* > 0) **then**
          begin *Print_String*(*TokenName*[*sy_Provided*]);
          Print_Proposition(*nSchemePremises*↑.*Items*↑[0]);
          for $i \leftarrow 1$ **to** *nSchemePremises*↑.*Count* − 1 **do**
            begin *Print_String*(*TokenName*[*sy_And*]); *Print_NewLine*;
            Print_Proposition(*nSchemePremises*↑.*Items*↑[*i*]);
            end;
          end;
        Print_String(*TokenName*[*sy_Proof*]); *Print_NewLine*;
        end;
    itTheorem: **with** *CompactStatementPtr*(*nContent*)↑ **do**
        begin *Print_NewLine*; *nIndent* ← 0; *Print_String*(*TokenName*[*sy_Theorem*]);
        Print_Label(*nProp*↑.*nLab*); *Print_NewLine*; *nIndent* ← 2; *Print_Indent*;
        Print_Formula(*nProp*↑.*nSentence*); *nIndent* ← 0; *Print_Justification*(*nJustification*, *nBlock*);
        Print_String(´;´); *Print_NewLine*;
        end;

```
itAxiom: begin end;
itReservation: with ReservationSegmentPtr(nContent)↑ do
    begin Print_NewLine; Print_String(TokenName[sy_reserve]);
    Print_Variable(nIdentifiers.Items↑[0]);
    for i ← 1 to nIdentifiers↑.Count − 1 do
      begin Print_String(´,´); Print_Variable(nIdentifiers↑.Items↑[i]);
      end;
    Print_String(TokenName[sy_For]); Print_ReservedType(nResType); Print_String(´;´);
    Print_NewLine;
    end;
itSection: begin Print_NewLine; Print_String(TokenName[sy_Begin]); Print_NewLine;
  end;
itRegularStatement: begin Print_RegularStatement(RegularStatementPtr(nContent), nBlock);
  Print_String(´;´); Print_NewLine;
  end;
itChoice: with ChoiceStatementPtr(nContent)↑ do
    begin if (nJustification↑.nInfSort = infStraightforwardJustification) ∧
          StraightforwardJustificationPtr(nJustification)↑.nLinked then
      begin Print_Linkage;
      end;
    Print_String(TokenName[sy_Consider]); Print_VariableSegment(nQualVars↑.Items↑[0]);
    for i ← 1 to nQualVars↑.Count − 1 do
      begin Print_String(´,´); Print_VariableSegment(nQualVars↑.Items↑[i]);
      end;
    if (nConditions ≠ nil) ∧ (nConditions↑.Count > 0) then
      begin Print_String(TokenName[sy_Such]); Print_Conditions(nConditions);
      end;
    Print_Justification(nJustification, nil); Print_String(´;´); Print_NewLine;
    end;
itReconsider: with TypeChangingStatementPtr(nContent)↑ do
    begin if (nJustification↑.nInfSort = infStraightforwardJustification) ∧
          StraightforwardJustificationPtr(nJustification)↑.nLinked then
      begin Print_Linkage;
      end;
    Print_String(TokenName[sy_Reconsider]);
    for i ← 0 to nTypeChangeList↑.Count − 1 do
      begin case TypeChangePtr(nTypeChangeList↑.Items↑[i])↑.nTypeChangeKind of
      Equating: begin Print_Variable(TypeChangePtr(nTypeChangeList↑.Items↑[i])↑.nVar);
        Print_String(´=´); Print_Term(TypeChangePtr(nTypeChangeList↑.Items↑[i])↑.nTermExpr);
        end;
      VariableIdentifier: begin Print_Variable(TypeChangePtr(nTypeChangeList.Items↑[i])↑.nVar);
        end;
      endcases;
      if (i ≥ 0) ∧ (i < nTypeChangeList↑.Count − 1) then Print_String(´,´);
      end;
    Print_String(TokenName[sy_As]); Print_Type(nTypeExpr);
    Print_Justification(nJustification, nil); Print_String(´;´); Print_NewLine;
    end;
itPrivFuncDefinition: with PrivateFunctorDefinitionPtr(nContent)↑ 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)↑ do
      begin Print_String(TokenName[sy_DefPred]); Print_Variable(nPredId); Print_String('[');
      Print_TypeList(nTypeExpList); Print_String(']'); Print_String(TokenName[sy_Means]);
      Print_Formula(nSentence); Print_String(';'); Print_NewLine;
      end;
  itConstantDefinition: with ConstantDefinitionPtr(nContent)↑ do
      begin Print_String(TokenName[sy_Set]); Print_Variable(nVarId); Print_String('=');
      Print_Term(nTermExpr); Print_String(';'); Print_NewLine;
      end;
  itLociDeclaration, itGeneralization: begin Print_String(TokenName[sy_Let]);
    Print_VariableSegment(QualifiedSegmentPtr(nContent)); Print_String(';'); Print_NewLine;
    end;
  itAssumption: begin Print_String(TokenName[sy_Assume]);
    Print_AssumptionConditions(AssumptionPtr(nContent)); Print_String(';'); Print_NewLine;
    end;
  itExistentialAssumption: with ExistentialAssumptionPtr(nContent)↑ do
      begin Print_String(TokenName[sy_Given]); Print_VariableSegment(nQVars↑.Items↑[0]);
      for i ← 1 to nQVars↑.Count − 1 do
        begin Print_String(','); Print_VariableSegment(nQVars↑.Items↑[i]);
        end;
      Print_String(TokenName[sy_Such]); Print_String(TokenName[sy_That]); Print_NewLine;
      Print_Proposition(nConditions↑.Items↑[0]);
      for i ← 1 to nConditions↑.Count − 1 do
        begin Print_String(TokenName[sy_And]); Print_NewLine;
        Print_Proposition(nConditions↑.Items↑[i]);
        end;
      Print_String(';'); Print_NewLine;
      end;
  itExemplification: with ExamplePtr(nContent)↑ do
      begin Print_String(TokenName[sy_Take]);
      if nVarId ≠ nil then
        begin Print_Variable(nVarId);
        if nTermExpr ≠ nil then
          begin Print_String('=');
          end;
        end;
      if nTermExpr ≠ nil then Print_Term(nTermExpr);
      Print_String(';'); Print_NewLine;
      end;
  itPerCases: begin if (JustificationPtr(nContent)↑.nInfSort =
          infStraightforwardJustification) ∧ StraightforwardJustificationPtr(nContent)↑.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;
```

```
    end;
  itCaseHead, itSupposeHead: begin Print_AssumptionConditions(AssumptionPtr(nContent));
    Print_String(´;´); Print_NewLine;
    end;
  itCorrCond: begin
        Print_String(CorrectnessName[CorrectnessConditionPtr(nContent)↑.nCorrCondSort]);
    Print_Justification(CorrectnessConditionPtr(nContent)↑.nJustification, nBlock); Print_String(´;´);
    Print_NewLine;
    end;
  itCorrectness: begin Print_String(TokenName[sy_Correctness]);
    Print_Justification(CorrectnessPtr(nContent)↑.nJustification, nBlock); Print_String(´;´);
    Print_NewLine;
    end;
  itProperty: begin Print_String(PropertyName[PropertyPtr(nContent)↑.nPropertySort]);
    Print_Justification(PropertyPtr(nContent)↑.nJustification, nBlock); Print_String(´;´);
    Print_NewLine;
    end;
  itDefMode: with ModeDefinitionPtr(nContent)↑ 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)↑.nExpansion);
        end;
      defStandardMode: with StandardModeDefinitionPtr(nContent)↑ do
          begin if nSpecification ≠ nil then
            begin Print_String(TokenName[sy_Arrow]); Print_Type(nSpecification);
            end;
          if nDefiniens ≠ nil then
            begin Print_String(TokenName[sy_Means]); Print_NewLine; Print_Definiens(nDefiniens);
            end;
          end;
      endcases; Print_String(´;´); Print_NewLine;
      end;
  itDefAttr: with AttributeDefinitionPtr(nContent)↑ 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)↑ do
      begin if nRedefinition then
        begin Print_String(TokenName[sy_Redefine]);
        end;
      Print_String(TokenName[sy_Pred]); Print_Pattern(nDefPredPattern);
      if nDefiniens ≠ nil then
        begin Print_String(TokenName[sy_Means]); Print_NewLine; Print_Definiens(nDefiniens);
        end;
```

```
      Print_String(´;´); Print_NewLine;
      end;
  itDefFunc: with FunctorDefinitionPtr(nContent)↑ do
      begin if nRedefinition then
        begin Print_String(TokenName[sy_Redefine]);
        end;
      Print_String(TokenName[sy_Func]); Print_Pattern(nDefFuncPattern);
      if nSpecification ≠ nil then
        begin Print_String(TokenName[sy_Arrow]); Print_Type(nSpecification);
        end;
      case nDefiningWay of
      dfEmpty: ;
      dfMeans: begin Print_String(TokenName[sy_Means]); Print_NewLine;
        end;
      dfEquals: begin Print_String(TokenName[sy_Equals]);
        end;
      endcases; Print_Definiens(nDefiniens); Print_String(´;´); Print_NewLine;
      end;
  itDefStruct: with StructureDefinitionPtr(nContent)↑ do
      begin Print_String(TokenName[sy_Struct]);
      if nAncestors↑.Count > 0 then
        begin Print_String(´(´); Print_Type(nAncestors↑.Items↑[0]);
        for i ← 1 to nAncestors↑.Count − 1 do
          begin Print_String(´,´); Print_Type(nAncestors↑.Items↑[i]);
          end;
        Print_String(´)´);
        end;
      Print_String(StructureName[nDefStructPattern↑.nModeSymbol]);
      if (nDefStructPattern↑.nArgs ≠ nil) ∧ (nDefStructPattern↑.nArgs↑.Count > 0) then
        begin Print_String(TokenName[sy_Over]); Print_Loci(nDefStructPattern↑.nArgs);
        end;
      Print_String(TokenName[sy_StructLeftBracket]);
      for i ← 0 to nSgmFields↑.Count − 1 do
        with FieldSegmentPtr(nSgmFields↑.Items↑[i])↑ do
          begin Print_String(SelectorName[FieldSymbolPtr(nFields↑.Items↑[0])↑.nFieldSymbol]);
          for j ← 1 to nFields↑.Count − 1 do
            with FieldSymbolPtr(nFields↑.Items↑[j])↑ do
              begin Print_String(´,´); Print_String(SelectorName[nFieldSymbol]);
              end;
          Print_String(TokenName[sy_Arrow]); Print_Type(nSpecification);
          if (i ≥ 0) ∧ (i < nSgmFields↑.Count − 1) then Print_String(´,´);
          end;
      Print_String(TokenName[sy_StructRightBracket]); Print_String(´;´); Print_NewLine;
      end;
  itPredSynonym, itFuncNotation, itModeNotation, itAttrSynonym:
          with NotationDeclarationPtr(nContent)↑ 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)↑ do
      begin Print_String(TokenName[sy_Antonym]); Print_Pattern(nNewPattern);
```

```
      Print_String(TokenName[sy_For]); Print_Pattern(nOriginPattern); Print_String(´;´);
      Print_NewLine;
      end;
  itCluster: begin Print_String(TokenName[sy_Cluster]);
    case ClusterPtr(nContent)↑.nClusterKind of
    ExistentialRegistration: with EClusterPtr(nContent)↑ do
        begin Print_AdjectiveList(nConsequent); Print_String(TokenName[sy_For]);
        Print_Type(nClusterType);
        end;
    ConditionalRegistration: with CClusterPtr(nContent)↑ 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)↑ do
        begin Print_Term(nClusterTerm); Print_String(TokenName[sy_Arrow]);
        Print_AdjectiveList(nConsequent);
        if nClusterType ≠ nil then
          begin Print_String(TokenName[sy_For]); Print_Type(nClusterType);
          end;
        end;
    endcases; Print_String(´;´); Print_NewLine;
    end;
  itIdentify: with IdentifyRegistrationPtr(nContent)↑ do
      begin Print_String(TokenName[sy_Identify]); Print_Pattern(nNewPattern);
      Print_String(TokenName[sy_With]); Print_Pattern(nOriginPattern);
      if (nEqLociList ≠ nil) ∧ (nEqLociList↑.Count > 0) then
        begin Print_String(TokenName[sy_When]);
        for i ← 0 to nEqLociList↑.Count − 1 do
          with LociEqualityPtr(nEqLociList↑.Items↑[i])↑ do
            begin Print_Locus(nLeftLocus); Print_String(´=´); Print_Locus(nRightLocus);
            if (i ≥ 0) ∧ (i < nEqLociList↑.Count − 1) then Print_String(´,´);
            end;
        end;
      Print_String(´;´); Print_NewLine;
      end;
  itPropertyRegistration: case PropertyRegistrationPtr(nContent)↑.nPropertySort of
    sySethood: with SethoodRegistrationPtr(nContent)↑ 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)↑ do
      begin Print_String(TokenName[sy_Reduce]); Print_Term(nOriginTerm);
      Print_String(TokenName[sy_To]); Print_Term(nNewTerm);
      end;
    Print_String(´;´); Print_NewLine;
    end;
  itPragma: begin Print_NewLine; Print_String(´::´ + PragmaPtr(nContent)↑.nPragmaStr);
    Print_NewLine;
    end;
```

itIncorrItem: ;
end;
endcases;
end;

1192. ⟨ Implementation for `wsmarticle.pas` 854 ⟩ +≡
procedure *Print_WSMizArticle*(*aWSTextProper* : *wsTextProperPtr*; *aFileName* : *string*);
var *lWSMizOutput*: *WSMizarPrinterPtr*;
begin *InitScannerNames*; *lWSMizOutput* ← *new*(*WSMizarPrinterPtr*, *OpenFile*(*aFileName*));
lWSMizOutput↑.*Print_TextProper*(*aWSTextProper*); *dispose*(*lWSMizOutput*, *Done*);
end;

File 22

Detour: Pragmas

1193. This chapter is a "detour" because it is out of order for the compiler, but it is a dependency for the next file (`parseradditions.pas`).

The `base/pragmas.pas` contains the global variables which are toggled by pragmas like "`::$P+`". This will toggle the *ProofPragma*. In particular, when *ProofPragma* is true, then Mizar will double check the proofs. When *ProofPragma* is false, Mizar will skip the proofs.

⟨ pragmas.pas 1193 ⟩ ≡
  ⟨ GNU License 4 ⟩
unit *pragmas*;
  interface uses *mobjects*;
  var *VerifyPragmaOn*, *VerifyPragmaOff*: *NatSet*; *VerifyPragmaIntervals*: *NatFunc*;
    SchemePragmaOn, *SchemePragmaOff*: *NatSet*; *SchemePragmaIntervals*: *NatFunc*;
    ProofPragma: *Boolean* = *true*; { check the proofs? }
  procedure *SetParserPragma*(*aPrg* : *string*);
  procedure *InsertPragma*(*aLine* : *integer*; *aPrg* : *string*);
  procedure *CompletePragmas*(*aLine* : *integer*);
  procedure *CanceledPragma* (**const** *aPrg*: *string*; **var** *aKind*: *char*; **var** *aNbr*: *integer*) ;
  implementation
  uses *mizenv*;

1194. Cancelling a definition or theorem is handled with the "`::$C`" pragma, which is administered only by the editors of the MML.

procedure *CanceledPragma* (**const** *aPrg*: *string*; **var** *aKind*: *char*; **var** *aNbr*: *integer*) ;
  var *lStr*: *string*; *k*, *lCod*: *integer*;
    begin *aKind* ← ´␣´;
    if (*Copy*(*aPrg*, 1, 2) = ´$C´) **then**
      begin if (*length*(*aPrg*) ≥ 3) ∧ (*aPrg*[3] ∈ [´D´, ´S´, ´T´]) **then**
        begin *aKind* ← *aPrg*[3]; *lStr* ← *TrimString*(*Copy*(*aPrg*, 4, *length*(*aPrg*) − 3)); *aNbr* ← 1;
        if *length*(*lStr*) > 0 **then**
          begin *k* ← 1;
          while (*k* ≤ *length*(*lStr*)) ∧ (*lStr*[*k*] ∈ [´0´ .. ´9´]) **do** *inc*(*k*);
          delete(*lStr*, *k*, *length*(*lStr*));
          if *length*(*lStr*) > 0 **then** *Val*(*lStr*, *aNbr*, *lCod*);
          end;
        end;
      end;
    end;

1195. 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*(*aPrg*, 1, 3) = ´`$P+`´ **then**
    begin *ProofPragma* ← *true*;
    end;
  if *copy*(*aPrg*, 1, 3) = ´`$P-`´ **then**
    begin *ProofPragma* ← *false*;
    end;
  end;

1196. The "`::$S+`" pragma will tell Mizar to check the scheme references, whereas "`::$S-`" pragma tells Mizar to stop verifying scheme references.

The "`::$V+`" pragma enables the verifier, and the "`::$V-`" pragma disables the verifier (skipping all verification until it is re-enabled).

procedure *InsertPragma*(*aLine* : *integer*; *aPrg* : *string*);
  begin if *copy*(*aPrg*, 1, 3) = ´`$V+`´ **then**
    begin *VerifyPragmaOn*.*InsertElem*(*aLine*); **end**;
  if *copy*(*aPrg*, 1, 3) = ´`$V-`´ **then**
    begin *VerifyPragmaOff*.*InsertElem*(*aLine*); **end**;
  if *copy*(*aPrg*, 1, 3) = ´`$S+`´ **then**
    begin *SchemePragmaOn*.*InsertElem*(*aLine*); **end**;
  if *copy*(*aPrg*, 1, 3) = ´`$S-`´ **then**
    begin *SchemePragmaOff*.*InsertElem*(*aLine*); **end**;
  end;

1197. 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*↑[i].X;
    for $j \leftarrow 0$ **to** *VerifyPragmaOn*.*Count* − 1 **do**
      begin $b \leftarrow$ *VerifyPragmaOn*.*Items*↑[j].X;
      if $b \geq a$ **then**
        begin *VerifyPragmaIntervals*.*Assign*(a, b); $f \leftarrow$ *true*; *break*; **end**;
      end;
    if $\neg f$ **then** *VerifyPragmaIntervals*.*Assign*(a, *aLine*);
    end;
  for $i \leftarrow 0$ **to** *SchemePragmaOff*.*Count* − 1 **do**
    begin $f \leftarrow$ *false*; $a \leftarrow$ *SchemePragmaOff*.*Items*↑[i].X;
    for $j \leftarrow 0$ **to** *SchemePragmaOn*.*Count* − 1 **do**
      begin $b \leftarrow$ *SchemePragmaOn*.*Items*↑[j].X;
      if $b \geq a$ **then**
        begin *SchemePragmaIntervals*.*Assign*(a, b); $f \leftarrow$ *true*; *break*; **end**;
      end;
    if $\neg f$ **then** *SchemePragmaIntervals*.*Assign*(a, *aLine*);
    end;
  end;

1198. Now we initialize the global variables declared in this module.

begin *VerifyPragmaOn.Init*(10, 10); *VerifyPragmaOff.Init*(10, 10);
VerifyPragmaIntervals.InitNatFunc(10, 10); *SchemePragmaOn.Init*(10, 10);
SchemePragmaOff.Init(10, 10); *SchemePragmaIntervals.InitNatFunc*(10, 10);
end.

File 23

Detour: Parser additions

1199. This chapter is a "detour" because we are "going out of [compiler] order" to discuss `parseradditions.pas`. Why? Well, because the file provides subclasses to those introduced in the abstract syntax unit, and are necessary for understanding the `parser.pas` unit.

One of the difficulties with this file is that there are 37 global variables declared here, and 46 module-wide variables, declared here. It's hard to juggle that knowledge! These "global" variables really describe the state of the parser, and do not seem to be used anywhere else.

⟦It would probably be wise to refactor the design to isolate these variables inside a `Parser` class, so they are not randomly distributed throughout this part of the program.⟧

CONVENTIONS: The classes have methods prefixed by *Start*, *Process*, and *Finish*.

- The *Start* methods reset the state variables needed to parse the syntactic entity.
- The *Process* methods usually update the state variables, either allocating new objects or transferring the current contents of a state variable in a different state variable.
- The *Finish* methods construct a WSM abstract syntax tree for the parsed entity.

⟨ parseraddition.pas 1199 ⟩ ≡
  ⟨ GNU License 4 ⟩
unit *parseraddition*;
  interface
  uses *syntax*, *errhan*, *mobjects*, *mscanner*, *abstract_syntax*, *wsmarticle*, *xml_inout*;
  procedure *InitWsMizarArticle*;
  type
    ⟨ Extended block class declaration 1205 ⟩
    ⟨ Extended item class declaration 1225 ⟩
    ⟨ Extended subexpression class declaration 1388 ⟩
    ⟨ Extended expression class declaration 1489 ⟩
  function *GetIdentifier*: *integer*;
  function *CreateArgs*(*aBase* : *integer*): *PList*;
  var ⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩
  implementation
  uses *mizenv*, *mconsole*, *parser*, *_formats*, *pragmas*
      mdebug , *info* **end_mdebug**;
  const *MaxSubTermNbr* = 64;
  var ⟨ Local variables for parser additions 1209 ⟩
    ⟨ Implementation of parser additions 1200 ⟩
  end .

1200. ⟨ Implementation of parser additions 1200 ⟩ ≡
⟨ Get the identifier number for current word 1201 ⟩
⟨ Initialize WS Mizar article 1203 ⟩;
⟨ Extended block implementation 1206 ⟩
⟨ Extended item implementation 1226 ⟩
⟨ Extended subexpression implementation 1390 ⟩
⟨ Extended expression implementation 1490 ⟩

This code is used in section 1199.

1201. When the current token is an identifier, we should obtain its number. If the current token is not an identifier, we should return 0.

⟨ Get the identifier number for current word 1201 ⟩ ≡
function *GetIdentifier*: *integer*;
  begin *result* ← 0;
  if *CurWord.Kind* = *Identifier* **then** *result* ← *CurWord.Nr*
  end;

This code is used in section 1200.

1202. Initializing a weakly-strict Mizar article requires setting the values for some of the global variables. Importantly, this will initialize the *gBlockPtr* in the Parser to be an *extBlockObj* instance. Note that this will create "the" *blMain* block object.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ ≡
gWSTextProper: *wsTextProperPtr*;
gLastWSBlock: *WSBlockPtr*;
gLastWSItem: *WSItemPtr*;

See also sections 1210, 1212, 1227, 1231, 1234, 1240, 1243, 1247, 1256, 1268, 1273, 1287, 1297, 1308, 1316, 1320, 1328, 1336, 1338, 1340, 1346, 1348, 1367, 1370, 1374, and 1394.

This code is used in section 1199.

1203. ⟨ Initialize WS Mizar article 1203 ⟩ ≡
procedure *InitWsMizarArticle*;
  begin { inintialize global variables which were declared in `parseraddition` }
  gWSTextProper ← *new*(*wsTextProperPtr*, *Init*(*ArticleID*, *ArticleExt*, *CurPos*));
  gLastWSBlock ← *gWSTextProper*; *gLastWSItem* ← **nil**;
  gBlockPtr ← *new*(*extBlockPtr*, *Init*(*blMain*)); { initialize other global variables }
  end;

This code is used in section 1200.

Section 23.1. EXTENDED BLOCK CLASS

1204. We extend the *Block* class (§691) introduced in the `syntax.pas` unit. Also recall the *wsBlock* class (§856) and the *wsItem* class (§860).

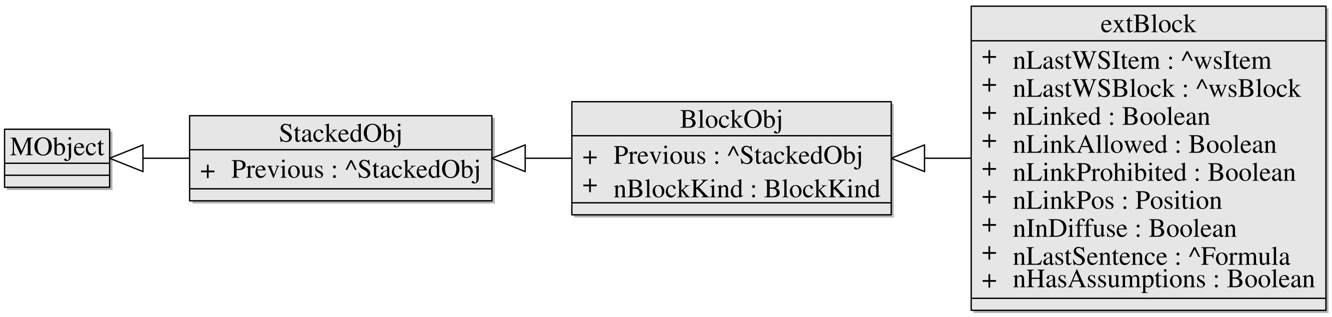

Fig. 6. Class hierarchy for *extBlockObj*, methods omitted.

1205. ⟨ Extended block class declaration 1205 ⟩ ≡

extBlockPtr = ↑*extBlockObj*;
extBlockObj = **object** (*BlockObj*)
  nLastWSItem: *WSItemPtr*;
  nLastWSBlock: *WSBlockPtr*;
  nLinked: *Boolean*; { is block prefixed by "`then`"? }
  nLinkAllowed: *Boolean*; { isn't this a duplicate of next field? }
  nLinkProhibited: *Boolean*; { can statement kind be prefixed by "`then`"? }
  nLinkPos: *Position*;
  nInDiffuse: *boolean*;
  nLastSentence: *FormulaPtr*;
  nHasAssumptions: *Boolean*;
  constructor *Init*(*fBlockKind* : *BlockKind*);
  procedure *Pop*; *virtual*;
  procedure *StartProperText*; *virtual*;
  procedure *ProcessRedefine*; *virtual*;
  procedure *ProcessLink*; *virtual*;
  procedure *ProcessBegin*; *virtual*;
  procedure *ProcessPragma*; *virtual*;
  procedure *StartSchemeDemonstration*; *virtual*;
  procedure *FinishSchemeDemonstration*; *virtual*;
  procedure *CreateItem*(*fItemKind* : *ItemKind*); *virtual*;
  procedure *CreateBlock*(*fBlockKind* : *BlockKind*); *virtual*;
end ;

This code is used in section 1199.

1206. Constructor. The constructor for an extended block object invokes the parent class's constructor (§694), initializes the instance variables, then its behaviour depends on whether we are constructing a "main" block or not.

⟨ Extended block implementation 1206 ⟩ ≡
constructor *extBlockObj.Init*(*fBlockKind* : *BlockKind*);
  begin *inherited Init*(*fBlockKind*);
  ⟨ Initialize default values for *extBlock* instance 1207 ⟩;
  if *nBlockKind* = *blMain* **then** ⟨ Initialize `main` *extBlock* instance 1208 ⟩
  else ⟨ Initialize "proper text" *extBlock* instance 1211 ⟩;
  end;

See also sections 1213, 1214, 1215, 1216, 1217, 1218, 1219, 1221, 1222, and 1223.
This code is used in section 1200.

1207. We have the default values suppose links are prohibited for the block, and there are no assumptions for the block. The last *wsItem* and *wsBlock* pointers are set to the global *gLastWSItem* and *gLastWSBlock* variables, respectively.

⟨ Initialize default values for *extBlock* instance 1207 ⟩ ≡
  nLinked ← *false*; *nLinkPos* ← *CurPos*; *nLinkAllowed* ← *false*; *nLinkProhibited* ← *true*;
  nHasAssumptions ← *false*; *gRedefinitions* ← *false*;
  nLastWSItem ← *gLastWSItem*; *nLastWSBlock* ← *gLastWSBlock*;

This code is used in section 1206.

1208. The "main" block of text needs to load the formats file, and populate the *gFormatsColl* (§648) and the *gFormatsBase* (*ibid.*) global variables. The `parseraddition.pas` unit's *gProofCnt* global variable is initialized to zero here.

⟨ Initialize `main` *extBlock* instance 1208 ⟩ ≡
  begin *nInDiffuse* ← *true*; *gProofCnt* ← 0;
  FileExam(*EnvFileName* + ´.frm´); *gFormatsColl.LoadFormats*(*EnvFileName* + ´.frm´);
  gFormatsBase ← *gFormatsColl.Count*; *setlength*(*Term*, *MaxSubTermNbr*);
  end

This code is used in section 1206.

1209. ⟨ Local variables for parser additions 1209 ⟩ ≡
Term: **array of** *TermPtr*; { (§753) }

See also sections 1228, 1232, 1238, 1241, 1244, 1245, 1248, 1252, 1258, 1260, 1262, 1269, 1271, 1274, 1278, 1284, 1292, 1298, 1302, 1309, 1317, 1331, 1382, and 1389.
This code is used in section 1199.

1210. ⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gProofCnt: *integer*;

1211. The “proper text” branch updates the *gLastWSBlock* global variable. For most of the kinds of blocks, we will have to toggle *nInDiffuse* to be true or false. For proof blocks, we will need to increment the “depth” counter tracking the proof block “nestedness”.

Only the “`case`” and “`suppose`” blocks, when determining if they are in “diffuse mode” or not, need to confer with the previous block. (Recall (§307), *StackedObj* classes has a *Previous* pointer.)

⟨ Initialize “proper text” *extBlock* instance 1211 ⟩ ≡

begin *gLastWSBlock* ← *gWsTextProper*↑.*NewBlock*(*nBlockKind*, *CurPos*);
mizassert(2341, *gLastWSItem* ≠ **nil**);
if *gLastWSItem*↑.*nItemKind* ∈ [*itDefinition*, *itRegularStatement*, *itSchemeBlock*, *itTheorem*, *itConclusion*, *itCaseBlock*, *itCorrCond*, *itCorrectness*, *itProperty*, *itPropertyRegistration*] **then**
wsItemPtr(*gLastWSItem*).*nBlock* ← *gLastWSBlock*;
case *nBlockKind* **of**
blDefinition: *nInDiffuse* ← *false*;
blNotation: *nInDiffuse* ← *false*;
blDiffuse: *nInDiffuse* ← *true*;
blHereby: *nInDiffuse* ← *true*;
blProof: **begin** *nLastSentence* ← *gLastFormula*; *inc*(*gProofCnt*); **end**;
blCase: *nInDiffuse* ← *extBlockPtr*(*Previous*)↑.*nInDiffuse*;
blSuppose: *nInDiffuse* ← *extBlockPtr*(*Previous*)↑.*nInDiffuse*;
blRegistration: *nInDiffuse* ← *false*;
blPublicScheme: *nInDiffuse* ← *false*;
endcases;
end

This code is used in section 1206.

1212. Popping a block. When we “pop” a proof block, we need to track the formula that was just proven and store it in the global variable *gLastFormula*.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gLastFormula: *FormulaPtr*;

1213. This actually implements the *Pop* method for blocks. When a block “closes” (i.e., the corresponding “`end`” statement has been encountered), we restore the global state’s *gLastWSItem* and *gLastWSBlock* pointers. When a proof block closes, we also restore the *gLastFormula* state.

Also note: the parent class’s method (§695) does nothing. This will be invoked in the *KillBlock* (§687).

⟨ Extended block implementation 1206 ⟩ +≡
procedure *extBlockObj*.*Pop*;
begin *gLastWSBlock*↑.*nBlockEndPos* ← *CurPos*;
case *nBlockKind* **of**
blProof: **begin** *gLastFormula* ← *nLastSentence*; *dec*(*gProofCnt*); **end**;
endcases;
gLastWSItem ← *nLastWSItem*; *gLastWSBlock* ← *nLastWSBlock*; { restore the “last” pointers }
inherited Pop;
end;

1214. Process "begin". Mizar uses "`begin`" to start a new "section" at the top-level of an article. Recall the grammar for this bit of Mizar:

```
Text-Proper = Section { Section } .
Section = "begin" { Text-Item } .
```

There are zero or more Text-Items in a section.

We should note that the main text is not organized as a linked list of "main" blocks. Instead, we have a single "main" block, and we just push an *itSection* item to its contents.

⟨ Extended block implementation 1206 ⟩ +≡
procedure *extBlockObj.ProcessBegin*;
  begin *nLinkAllowed* ← *false*; *nLinkProhibited* ← *true*;
  gLastWSItem ← *gWsTextProper*↑.*NewItem*(*itSection*, *CurPos*); *nLastWSItem* ← *gLastWSItem*;
  gLastWSBlock↑.*nItems*.*Insert*(*gLastWSItem*);
  end;

1215. This will add a pragma item to the current block. The parser's *ProcessPragmas* (§1585) invokes this method.

⟨ Extended block implementation 1206 ⟩ +≡
procedure *extBlockObj.ProcessPragma*;
  begin *nLinkAllowed* ← *false*; *nLinkProhibited* ← *true*;
    { Create a new item }
  gLastWSItem ← *gWsTextProper*↑.*NewItem*(*itPragma*, *CurPos*);
  gLastWSItem↑.*nContent* ← *new*(*PragmaPtr*, *Init*(*CurWord*.*Spelling*));
    { Insert the pragma, update last item in block }
  nLastWSItem ← *gLastWSItem*; *gLastWSBlock*↑.*nItems*.*Insert*(*gLastWSItem*);
  end;

1216. Starting the proper text will just update the *nBlockPos* field to whatever the current position is.

⟨ Extended block implementation 1206 ⟩ +≡
procedure *extBlockObj.StartProperText*;
  begin *gWSTextProper*↑.*nBlockPos* ← *CurPos*; **end**;

1217. Processing redefinitions sets the global variable *gRedefinitions* to the result of comparing the current word to the "`redefine`" keyword.

⟨ Extended block implementation 1206 ⟩ +≡
procedure *extBlockObj.ProcessRedefine*;
  begin *gRedefinitions* ← *CurWord*.*Kind* = *sy_Redefine*; **end**;

1218. When a block statement is linked, but it should not, then we raise a `164` error. Otherwise, be sure to mark the block as linked (i.e., toggle *nLinked* to be true) and assign the *nLinkPos* to be the current position.

⟨ Extended block implementation 1206 ⟩ +≡
procedure *extBlockObj.ProcessLink*;
  begin if *CurWord*.*Kind* ∈ [*sy_Then*, *sy_Hence*] **then**
    begin if *nLinkProhibited* **then** *ErrImm*(164);
    nLinked ← *true*; *nLinkPos* ← *CurPos*;
    end;
  end;

1219. Proof of a scheme. We should increment the proof depth global variable.

Recall that *ProofPragma* means "check the proof is valid?" In other words, when *ProofPragma* is false, we are skipping the proofs.

§thesis-formula:macro-def

define *thesis_formula* ≡ *new*(*ThesisFormulaPtr*, *Init*(*CurPos*))
define *thesis_prop* ≡ *new*(*PropositionPtr*, *Init*(*new*(*LabelPtr*, *Init*(0, *CurPos*)), *thesis_formula*, *CurPos*))
define *skipped_proof_justification* ≡ *new*(*JustificationPtr*, *Init*(*infSkippedProof*, *CurPos*))

⟨ Extended block implementation 1206 ⟩ +≡
procedure *extBlockObj.StartSchemeDemonstration*;
  begin *inc*(*gProofCnt*);
  if ¬*ProofPragma* **then** ⟨ Mark schema proof as "skipped" 1220 ⟩;
  end;

1220. When we skip the proof (due to pragmas being set), we just add the scheme as a compact statement whose justification is the "skipped proof justification".

First, we create a new text item for the proper text global variable. Then we set its content to the compact statement with the "skipped" justification. Finally we add this item to the "last" (latest) *wsBlock* global variable.

⟨ Mark schema proof as "skipped" 1220 ⟩ ≡
  begin *gLastWSItem* ← *gWsTextProper*↑.*NewItem*(*itConclusion*, *CurPos*);
  gLastWSItem↑.*nContent* ← *new*(*CompactStatementPtr*, *Init*(*thesis_prop*, *skipped_proof_justification*));
  gLastWSBlock↑.*nItems*.*Insert*(*gLastWSItem*);
  end

This code is used in section 1219.

1221. Finishing the proof for a scheme should decrement the global "proof depth" counter.

⟨ Extended block implementation 1206 ⟩ +≡
procedure *extBlockObj.FinishSchemeDemonstration*;
  begin *dec*(*gProofCnt*); **end**;

1222. The factory method for *extBlock* creating an item will update the global *gItemPtr* variable (§690).

⟨ Extended block implementation 1206 ⟩ +≡
procedure *extBlockObj.CreateItem*(*fItemKind* : *ItemKind*);
  begin *gItemPtr* ← *new*(*extItemPtr*, *Init*(*fItemKind*)); **end**;

1223. The factory method for *extBlock* creating a new block will update the *gBlockPtr* global variable (§690).

⟨ Extended block implementation 1206 ⟩ +≡
procedure *extBlockObj.CreateBlock*(*fBlockKind* : *BlockKind*);
  begin *gBlockPtr* ← *new*(*extBlockPtr*, *Init*(*fBlockKind*)) **end**;

Section 23.2. EXTENDED ITEM CLASS

1224. The class diagram for extended items looks like:

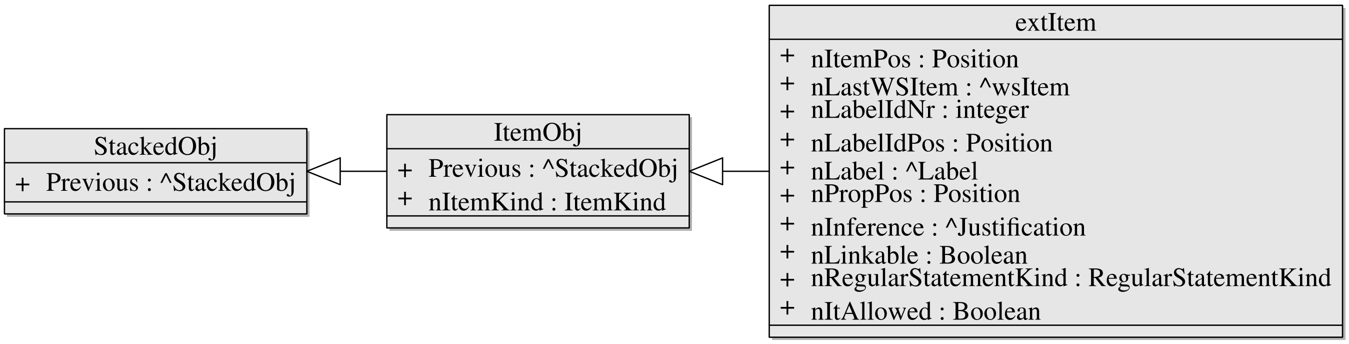

Fig. 7. Class hierarchy for *extItemObj*. The base *MObject* class omitted from the hierarchy.

Recall (§909) the regular statement kind is one of three possibilities: diffuse statement, compact statement, iterative equality.

The "Finish" methods updates the contents of the *extItem* class with a WSM abstract syntax tree for the statement.

Since this is a "stub", I will just leave the placeholder chunk for the methods overriden by the extended Item class here (remove later).

⟨Methods overriden by extended Item class 706⟩ +≡

1225. ⟨Extended item class declaration 1225⟩ ≡

extItemPtr = ↑*extItemObj*;
extItemObj = **object** (*ItemObj*)
  nItemPos: *Position*;
  nLastWSItem: *WSItemPtr*;
  nLabelIdNr: *integer*;
  nLabelIdPos: *Position*;
  nLabel: *LabelPtr*;
  nPropPos: *Position*;
  nInference: *JustificationPtr*;
  nLinkable: *boolean*;
  nRegularStatementKind: *RegularStatementKind*;
  nItAllowed: *boolean*;
  constructor *Init*(*fKind* : *ItemKind*);
  procedure *Pop*; *virtual*;
  ⟨Methods overriden by extended Item class 706⟩
end ;

This code is used in section 1199.

Subsection 23.2.1. Constructor

1226. There are a number of comments in Polish which I haphazardly translated into English (“Przygotowanie definiensow:” translates as “Preparation of definiens:”; “Ew. zakaz przy obiektach ekspandowanych” translates as “Possible ban on expanded facilities”)

⟨ Extended item implementation 1226 ⟩ ≡
constructor *extItemObj.Init*(*fKind* : *ItemKind*);
  begin *inherited Init*(*fKind*);
  ⟨ Initialize the fields for newly allocated *extItem* object 1229 ⟩
  mizassert(2343, $gLastWSBlock \neq$ **nil**);
  if $\neg$(*nItemKind* $\in$ [*itReservation*, *itConstantDefinition*, *itExemplification*, *itGeneralization*,
      itLociDeclaration]) **then**
    begin *gLastWSItem* $\leftarrow$ *gWsTextProper*↑.*NewItem*(*fKind*, *CurPos*); *nLastWSItem* $\leftarrow$ *gLastWSItem*;
    end;
  case *nItemKind* **of**
    ⟨ Initialize extended item by *ItemKind* 1230 ⟩
  endcases;
  if $\neg$(*nItemKind* $\in$ [*itReservation*, *itConstantDefinition*, *itExemplification*, *itGeneralization*,
      itLociDeclaration]) **then** *gLastWSBlock*↑.*nItems*.*Insert*(*gLastWSItem*);
  end;

See also sections 1249, 1277, 1279, 1280, 1281, 1282, 1283, 1285, 1286, 1288, 1289, 1290, 1291, 1293, 1294, 1295, 1296, 1299, 1300, 1301, 1303, 1304, 1305, 1306, 1307, 1310, 1311, 1312, 1313, 1314, 1315, 1318, 1319, 1321, 1322, 1323, 1324, 1325, 1326, 1327, 1329, 1330, 1332, 1333, 1334, 1335, 1337, 1339, 1341, 1342, 1343, 1344, 1345, 1347, 1349, 1350, 1351, 1352, 1353, 1354, 1355, 1356, 1357, 1358, 1359, 1360, 1361, 1362, 1363, 1364, 1365, 1366, 1368, 1369, 1371, 1372, 1373, 1375, 1376, 1377, 1378, 1379, 1380, 1381, 1383, 1384, and 1385.

This code is used in section 1200.

1227. Initializing the fields. The *it_Allowed* global variable is toggled on and off when the parser encounters “guards” in conditional definitions, whereas the *nItAllowed* fields reflects whether the sort of definition allows “`it`” in the definiens.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
dol_Allowed: *Boolean*;
it_Allowed: *Boolean*;
in_AggrPattern: *Boolean*;

gLastType: *TypePtr*;
gLastTerm: *TermPtr*;
gDefiningWay: *HowToDefine*;

1228. ⟨ Local variables for parser additions 1209 ⟩ +≡
gClusterSort: *ClusterRegistrationKind*;
gDefiniens: *DefiniensPtr*;
gPartialDefs: *PList*;
nDefiniensProhibited: *boolean*;
gSpecification: *TypePtr*;

1229. ⟨ Initialize the fields for newly allocated *extItem* object 1229 ⟩ ≡
nItemPos ← *CurPos*; *gClusterSort* ← *ExistentialRegistration*; *nItAllowed* ← *false*; *it_Allowed* ← *false*; { global variable! }
in_AggrPattern ← *false*; *dol_Allowed* ← *false*; *gSpecification* ← **nil**; *gLastType* ← **nil**;
gLastFormula ← **nil**; *gLastTerm* ← **nil**;
{ Preparation of definiens: }
nDefiniensProhibited ← *false*;
{ Possible ban on expanded facilities }
gDefiningWay ← *dfEmpty*; *gDefiniens* ← **nil**; *gPartialDefs* ← **nil**; *nLinkable* ← *false*;

This code is used in section 1226.

1230. Kind-specific initialization. Each kind of item may need some specific initialization. We work through all the cases. The first two cases considered are generalization ("`let` ⟨*Qualified Variables*⟩ `be` [`such` ⟨*Conditions*⟩]") and existential assumptions ("`given` ⟨*Qualified Variables*⟩ `such` ⟨ *Conditions*⟩"). Existential assumptions need to toggle the "has assumptions" field to true for the global block pointer.

⟨ Initialize extended item by *ItemKind* 1230 ⟩ ≡
itGeneralization: ; { `let` statements }
itExistentialAssumption: *ExtBlockPtr*(*gBlockPtr*)↑.*nHasAssumptions* ← *true*;

See also sections 1233, 1235, 1237, 1239, 1242, and 1246.

This code is used in sections 1226 and 1244.

1231. Property initialization. Initializing a property statement *Item* should raise an error when the property does not appear in the correct block.

• Defining a predicate can support the following properties: symmetry, reflectivity, irreflexivity, transitivity, conectedness, asymmetry.
• Functors can support: associativity, commutativity, idempotence, involutiveness, and projectivity properties.
• Modes can support the sethood property.

In all other situations, an error should be flagged (the user is trying to assert an invalid property).

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gDefKind: *ItemKind*;

1232. ⟨ Local variables for parser additions 1209 ⟩ +≡
gExpandable: *boolean*;
gPropertySort: *PropertyKind*;

1233. ⟨ Initialize extended item by *ItemKind* 1230 ⟩ +≡
itProperty: **begin** *gPropertySort* ← *PropertyKind*(*CurWord*.*Nr*);
  case *PropertyKind*(*CurWord*.*Nr*) **of**
  sySymmetry, *syReflexivity*, *syIrreflexivity*, *syTransitivity*, *syConnectedness*, *syAsymmetry*:
    if $gDefKind \neq itDefPred$ **then**
      begin *ErrImm*(81); *gPropertySort* ← *sErrProperty*; **end**;
  syAssociativity, *syCommutativity*, *syIdempotence*: **if** $gDefKind \neq itDefFunc$ **then**
      begin *ErrImm*(82); *gPropertySort* ← *sErrProperty*; **end**;
  syInvolutiveness, *syProjectivity*: **if** $gDefKind \neq itDefFunc$ **then**
      begin *ErrImm*(83); *gPropertySort* ← *sErrProperty*; **end**;
  sySethood: **if** $(gDefKind \neq itDefMode) \lor gExpandable$ **then**
      begin *ErrImm*(86); *gPropertySort* ← *sErrProperty*; **end**;
  endcases;
  end;

1234. Reconsider initialization. We need to allocate a new (empty) list for the list of terms being reconsidered.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gReconsiderList: *PList*;

1235. ⟨ Initialize extended item by *ItemKind* 1230 ⟩ +≡
itReconsider: *gReconsiderList* ← *new*(*PList*, *Init*(0));

1236. We can have in Mizar "`suppose that` ⟨*statement*⟩" (as well as "`case that`..."). But in those cases, the statement cannot be linked to the next statement (i.e., the next statement cannot begin with "`then`..."). Assumptions without "`that`" are always linkable.r

Theorems, "regular statements", and conclusions are always linkable.

1237. ⟨ Initialize extended item by *ItemKind* 1230 ⟩ +≡
itRegularStatement: *nLinkable* ← *true*;
itConclusion: *nLinkable* ← *true*;
itPerCases: ;
itCaseHead: **if** *AheadWord.Kind* ≠ *sy_That* **then** *nLinkable* ← *true*;
itSupposeHead: **if** *AheadWord.Kind* ≠ *sy_That* **then** *nLinkable* ← *true*;
itTheorem: *nLinkable* ← *true*;
itAxiom: **if** ¬*AxiomsAllowed* **then** *ErrImm*(66);
itChoice: ;

1238. Initializing an assumption. Collective assumptions ("`assume that` ⟨*formula*⟩") are not linkable, but single assumptions ("`assume` ⟨*Proposition*⟩") are linkable. The statement will introduce a list of premises, which will be tracked in the *gPremises* local variable for the module.

⟨ Local variables for parser additions 1209 ⟩ +≡
gPremises: *PList*;

1239. ⟨ Initialize extended item by *ItemKind* 1230 ⟩ +≡
itAssumption: **begin if** *AheadWord.Kind* ≠ *sy_That* **then** *nLinkable* ← *true*;
  gPremises ← **nil**;
  end;

1240. Definition items. Definition items need to be initialized with some nuance. Some definitions permit "`it`" to be used in the definiens, but others do not. Mizar toggles the global variables tracking this here. There is a common set of things toggled which we have isolated as the `WEB` macro *initialize_definition_item* common to initializing all definition items.

The correctness conditions are determined at this point, as well.

define *initialize_definition_item* ≡ *gCorrectnessConditions* ← [ ]; *gDefPos* ← *CurPos*;
    gDefKind ← *nItemKind*

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gCorrectnessConditions: *CorrectnessConditionsSet*;

1241. ⟨ Local variables for parser additions 1209 ⟩ +≡
gDefPos: *Position*;
gStructPrefixes: *PList*;

1242. ⟨ Initialize extended item by *ItemKind* 1230 ⟩ +≡
itLociDeclaration: ;
itDefMode: **begin** *nItAllowed* ← *true*; *gExpandable* ← *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* ← *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*; *gPropertySort* ← *PropertyKind*(*CurWord.Nr*);
  end;
itDefStruct: **begin** *initialize_definition_item*; *gStructPrefixes* ← *new*(*PList*, *Init*(0)); **end**;
itCanceled: **begin** *ErrImm*(88); **end**;

1243. Correctness conditions. Registrations and definitions need correctness conditions to ensure the well-definedness of adjective clusters and terms. The correctness conditions needed for a definition (or registration) are inserted into the *gCorrectnessConditions* variable. When the correctness condition is found, we remove it from the *gCorrectnessConditions* set.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gRedefinitions: *boolean*;

1244. ⟨ Local variables for parser additions 1209 ⟩ +≡
gCorrCondSort: *CorrectnessKind*;
⟨ Initialize extended item by *ItemKind* 1230 ⟩ = *itCorrCond*:
    if *CorrectnessKind*(*CurWord.Nr*) ∈ *gCorrectnessConditions* **then**
  begin *exclude*(*gCorrectnessConditions*, *CorrectnessKind*(*CurWord.Nr*));
  gCorrCondSort ← *CorrectnessKind*(*CurWord.Nr*);
  if (*gRedefinitions* ∧ (*gCorrCondSort* = *syCoherence*) ∧ *ExtBlockPtr*(*gBlockPtr*)↑.*nHasAssumptions*)
    then *ErrImm*(243);
  end
else begin *ErrImm*(72); *gCorrCondSort* ← *CorrectnessKind*(0); **end**;
itCorrectness: **if** (*gRedefinitions* ∧ *ExtBlockPtr*(*gBlockPtr*)↑.*nHasAssumptions*) **then** *ErrImm*(243);

1245. The last statement needing attention will be the `scheme` block. Note that *gLocalScheme* is not used anywhere.

⟨ Local variables for parser additions 1209 ⟩ +≡
gLocalScheme: *boolean*;
gSchemePos: *Position*;

1246. ⟨ Initialize extended item by *ItemKind* 1230 ⟩ +≡
itDefinition, *itSchemeHead*, *itReservation*, *itPrivFuncDefinition*, *itPrivPredDefinition*, *itConstantDefinition*,
  itExemplification: ;
itCaseBlock: ;
itSchemeBlock: **begin** *gLocalScheme* ← *CurWord.Kind* ≠ *sy_Scheme*; *gSchemePos* ← *CurPos*; **end**;

1247. Popping an extended item.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gSchemeParams: *PList*;

1248. ⟨ Local variables for parser additions 1209 ⟩ +≡
gPatternPos: *Position*;
gPattern: *PatternPtr*;
gNewPatternPos: *Position*;
gNewPattern: *PatternPtr*;
gSchemeIdNr: *integer*;
gSchemeIdPos: *Position*;
gSchemeConclusion: *FormulaPtr*;
gSchemePremises: *PList*;

Subsection 23.2.2. Popping

1249. Popping an item is invoked as part of *KillItem*, which occurs whenever (1) a semicolon is encountered, or (2) when starting a proof environment.

The contract for popping an item ensures the *nContent* field shall be populated for valid items.

NOTE: PASCAL has a set operation *include*(*set*, *element*) which adjoins an *element* to a *set*.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.Pop*;
  var *k*: *integer*;
  begin *gLastWSItem*↑.*nItemEndPos* ← *PrevPos*; ⟨ Check for errors with definition items 1253 ⟩
  ⟨ Update content of *nLastWSItem* based on type of item popped 1250 ⟩;
  ⟨ Check the popped item's linkages are valid 1276 ⟩;
  if *gDefiningWay* ≠ *dfEmpty* **then**
    begin if *gDefiniens*↑.*nDefSort* = *ConditionalDefiniens* **then**
      include(*gCorrectnessConditions*, *syConsistency*);
    if *gRedefinitions* **then** *include*(*gCorrectnessConditions*, *syCompatibility*);
    end;
  inherited Pop; { (§704) }
  end;

1250. We will update the caller's *nLastWSItem*'s contents in most cases.

⟨ Update content of *nLastWSItem* based on type of item popped 1250 ⟩ ≡
case *nItemKind* **of**
itTheorem: *nLastWSItem*↑.*nContent* ← *new*(*CompactStatementPtr*, *Init*(*new*(*PropositionPtr*, *Init*(*nLabel*, *gLastFormula*, *nPropPos*)), *nInference*));
⟨ Pop a proof step 1254 ⟩
itConclusion, *itRegularStatement*: ⟨ Pop a conclusion or regular statement 1261 ⟩
itGeneralization, *itLociDeclaration*: ⟨ Pop a "`let`" statement 1263 ⟩
⟨ Pop a definition item 1264 ⟩
itPredSynonym, *itPredAntonym*, *itFuncNotation*, *itModeNotation*, *itAttrSynonym*, *itAttrAntonym*: *nLastWSItem*↑.*nContent* ← *new*(*NotationDeclarationPtr*, *Init*(*gNewPatternPos*, *nItemKind*, *gNewPattern*, *gPattern*));
⟨ Pop a registration item 1272 ⟩
itCorrCond: *nLastWSItem*↑.*nContent* ← *new*(*CorrectnessConditionPtr*, *Init*(*nItemPos*, *gCorrCondSort*, *nInference*));
itCorrectness: *nLastWSItem*↑.*nContent* ← *new*(*CorrectnessConditionsPtr*, *Init*(*nItemPos*, *gCorrectnessConditions*, *nInference*));
itProperty: *nLastWSItem*↑.*nContent* ← *new*(*PropertyPtr*, *Init*(*nItemPos*, *gPropertySort*, *nInference*));
itSchemeHead: *nLastWSItem*↑.*nContent* ← *new*(*SchemePtr*, *Init*(*gSchemeIdNr*, *gSchemeIdPos*, *gSchemeParams*, *gSchemePremises*, *gSchemeConclusion*));
⟨ Pop **skip**s remaining cases 1251 ⟩
endcases

This code is used in section 1249.

1251. ⟨ Pop **skip**s remaining cases 1251 ⟩ ≡
itPrivFuncDefinition, *itPrivPredDefinition*, *itPragma*, *itDefinition*, *itSchemeBlock*, *itReservation*, *itExemplification*, *itCaseBlock*: ;

This code is used in section 1250.

1252. 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).

⟨ Local variables for parser additions 1209 ⟩ +≡
gMeansPos: *Position*;

1253. ⟨ Check for errors with definition items 1253 ⟩ ≡
case *nItemKind* **of**
itDefPred, *itDefFunc*, *itDefMode*, *itDefAttr*: **begin if** *gDefiningWay* ≠ *dfEmpty* **then**
begin if *nDefiniensProhibited* ∧ ¬*AxiomsAllowed* **then**
begin *Error*(*gMeansPos*, 254); *gDefiningWay* ← *dfEmpty*; **end**;
end
else if ¬*gRedefinitions* ∧ ¬*nDefiniensProhibited* ∧ ¬*AxiomsAllowed* **then** *SemErr*(253);
end;
endcases;

This code is used in section 1249.

1254. Pop a proof step. Popping a proof step should assign to the contents of the caller's *nLastWsItem* some kind of inference justification, usually in the form of a statement in the WSM syntax tree.

⟨ Pop a proof step 1254 ⟩ ≡
itPerCases: *nLastWSItem*↑.*nContent* ← *nInference*;

See also sections 1255, 1257, and 1259.
This code is used in section 1250.

1255. Popping a reconsideration. We should assign a *TypeChangingStatement* to the content of the caller's last item, using the *nInference* field of the caller as the justification.

⟨ Pop a proof step 1254 ⟩ +≡
itReconsider: *nLastWSItem*↑.*nContent* ← *new*(*TypeChangingStatementPtr*, *Init*(*gReconsiderList*, *gLastType*, *SimpleJustificationPtr*(*nInference*)));

1256. Popping existential elimination and introduction. We assign a `consider` (or `given`) WSM statement to the caller's previous *WSItem*'s contents when popping a choice (resp., existential assumption) item.

We should remind the reader of the grammar here:

⟨*Qualified-Segment*⟩ ::= ⟨*Variables*⟩ ⟨*Qualification*⟩
⟨*Variables*⟩ ::= ⟨*Variable*⟩ { "," ⟨*Variable*⟩ }
⟨*Qualification*⟩ ::= ("being" | "be") ⟨*Type-Expression*⟩

And, of course, a qualified-segment list is just a comma-separated list of qualified-segments.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gQualifiedSegmentList: *PList*;

1257. ⟨ Pop a proof step 1254 ⟩ +≡
itChoice: **begin** *nLastWSItem*↑.*nContent* ← *new*(*ChoiceStatementPtr*, *Init*(*gQualifiedSegmentList*, *gPremises*, *SimpleJustificationPtr*(*nInference*))); *gPremises* ← **nil**;
end;
itExistentialAssumption: **begin** *nLastWSItem*↑.*nContent* ← *new*(*ExistentialAssumptionPtr*, *Init*(*nItemPos*, *gQualifiedSegmentList*, *gPremises*)); *gPremises* ← **nil**;
end;

1258. Popping a stipulation. When we pop a `case`, `suppose`, or `assume` — some kind of "assumption"-like statement — we are assigning either a *CollectiveAssumption* object or a *SingleAssumption* object to the content of the *current WSItem* **global** variable.

⟨ Local variables for parser additions 1209 ⟩ +≡
gThatPos: *Position*;

1259. ⟨ Pop a proof step 1254 ⟩ +≡
itSupposeHead, *itCaseHead*, *itAssumption*: **if** *gPremises* ≠ **nil then**
begin *gLastWSItem*↑.*nContent* ← *new*(*CollectiveAssumptionPtr*, *Init*(*gThatPos*, *gPremises*));
gPremises ← **nil**;
end
else *gLastWSItem*↑.*nContent* ← *new*(*SingleAssumptionPtr*, *Init*(*nItemPos*, *new*(*PropositionPtr*, *Init*(*nLabel*, *gLastFormula*, *nPropPos*))));

1260. Pop a conclusion or regular statement. We assign an appropriate WSM statement node to the previous item's contents.

⟨ Local variables for parser additions 1209 ⟩ +≡
gIterativeSteps: *PList*;
gIterativeLastFormula: *FormulaPtr*;
gInference: *JustificationPtr*;

1261. ⟨ Pop a conclusion or regular statement 1261 ⟩ ≡
case *nRegularStatementKind* **of**
stDiffuseStatement:
nLastWSItem↑.*nContent* ← *new*(*DiffuseStatementPtr*, *Init*(*nLabel*, *stDiffuseStatement*));
stCompactStatement: *nLastWSItem*↑.*nContent* ← *new*(*CompactStatementPtr*, *Init*(*new*(*PropositionPtr*, *Init*(*nLabel*, *gLastFormula*, *nPropPos*)), *nInference*));
stIterativeEquality: *nLastWSItem*↑.*nContent* ← *new*(*IterativeEqualityPtr*, *Init*(*new*(*PropositionPtr*, *Init*(*nLabel*, *gIterativeLastFormula*, *nPropPos*)), *gInference*, *gIterativeSteps*));
endcases;

This code is used in section 1250.

1262. Pop a 'let' statement. For generic let statements of the form

$$\texttt{let}\ \vec{x}_1\ \texttt{be}\ T_1, \ldots, \vec{x}_n\ \texttt{be}\ T_n$$

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

$$\texttt{let}\ \vec{x}\ \texttt{be}\ T\ \texttt{such that}\ \Phi$$

we need to add a *CollectiveAssumption* node to the **global** *gLastWSBlock*'s items.

⟨ Local variables for parser additions 1209 ⟩ +≡
gSuchPos: *Position*;

1263. ⟨ Pop a "let" statement 1263 ⟩ ≡
begin for $k \leftarrow 0$ **to** *gQualifiedSegmentList*↑.*Count* − 1 **do**
begin *gLastWSItem* ← *gWsTextProper*↑.*NewItem*(*nItemKind*, *QualifiedSegmentPtr*(*gQualifiedSegmentList*↑.*Items*↑[*k*])↑.*nSegmPos*);
nLastWSItem ← *gLastWSItem*; *gLastWSItem*↑.*nContent* ← *gQualifiedSegmentList*↑.*Items*↑[*k*];
if $k =$ *gQualifiedSegmentList*↑.*Count* − 1 **then** *gLastWSItem*↑.*nItemEndPos* ← *PrevPos*
else *gLastWSItem*↑.*nItemEndPos* ← *QualifiedSegmentPtr*(*gQualifiedSegmentList*↑.*Items*↑[*k* + 1])↑.*nSegmPos*;
gQualifiedSegmentList↑.*Items*↑[*k*] ← **nil**; *gLastWSBlock*↑.*nItems*.*Insert*(*gLastWSItem*);
end;
dispose(*gQualifiedSegmentList*, *Done*);
if *gPremises* ≠ **nil then**
begin *gLastWSItem* ← *gWsTextProper*↑.*NewItem*(*itAssumption*, *gSuchPos*);
gLastWSItem↑.*nContent* ← *new*(*CollectiveAssumptionPtr*, *Init*(*gThatPos*, *gPremises*));
gPremises ← **nil**; *gLastWSItem*↑.*nItemEndPos* ← *PrevPos*; *nLastWSItem* ← *gLastWSItem*;
gLastWSBlock↑.*nItems*.*Insert*(*gLastWSItem*);
end;
end;

This code is used in section 1250.

1264. Pop a mode definition. A mode is either expandable (an abbreviation) or nonexpandable. For expandable modes, we just add a new *ExpandableModeDefinition* WSM object to the caller's *nLastWSItem*'s contents.

On the other hand, non-expandable modes should add to the caller's *nLastWSItem*'s contents a new *StandardModeDefinition* object. If this is not a redefinition, then we must add the "`existence`" correctness condition to the global variable *gCorrectnessConditions*.

⟨ Pop a definition item 1264 ⟩ ≡
itDefMode: **begin if** *gExpandable* **then** *nLastWSItem*↑.*nContent* ← *new*(*ExpandableModeDefinitionPtr*, *Init*(*gPatternPos*, *ModePatternPtr*(*gPattern*), *gLastType*))
else begin *nLastWSItem*↑.*nContent* ← *new*(*StandardModeDefinitionPtr*, *Init*(*gPatternPos*, *gRedefinitions*, *ModePatternPtr*(*gPattern*), *gSpecification*, *gDefiniens*));
if ¬*gRedefinitions* **then** *include*(*gCorrectnessConditions*, *syExistence*);
end;
end;

See also sections 1265, 1266, 1267, and 1270.

This code is used in section 1250.

1265. Pop a functor definition. When popping a functor definition, we just add a *FunctorDefinition* object to the caller's *nLastWSItem*'s contents.

⟨ Pop a definition item 1264 ⟩ +≡
itDefFunc: **begin** *nLastWSItem*↑.*nContent* ← *new*(*FunctorDefinitionPtr*, *Init*(*gPatternPos*, *gRedefinitions*, *FunctorPatternPtr*(*gPattern*), *gSpecification*, *gDefiningWay*, *gDefiniens*));
end;

1266. Pop an attribute definition. We just need to add an *AttributeDefinition* object to the caller's *nLastWSItem*'s contents.

⟨ Pop a definition item 1264 ⟩ +≡
itDefAttr: **begin** *nLastWSItem*↑.*nContent* ← *new*(*AttributeDefinitionPtr*, *Init*(*gPatternPos*, *gRedefinitions*, *AttributePatternPtr*(*gPattern*), *gDefiniens*));
end;

1267. Pop a predicate definition. We just need to add a *PredicateDefinition* object to the caller's *nLastWSItem*'s contents.

⟨ Pop a definition item 1264 ⟩ +≡
itDefPred: **begin** *nLastWSItem*↑.*nContent* ← *new*(*PredicateDefinitionPtr*, *Init*(*gPatternPos*, *gRedefinitions*, *PredicatePatternPtr*(*gPattern*), *gDefiniens*));
end;

1268. Popping a structure definition. We just need to add a *StructureDefinition* object to the caller's *nLastWSItem*'s contents.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gConstructorNr: *integer*;

1269. ⟨ Local variables for parser additions 1209 ⟩ +≡
gParams: *PList*;
gStructFields: *PList*;

1270. ⟨ Pop a definition item 1264 ⟩ +≡
itDefStruct: **begin** *nLastWSItem*↑.*nContent* ← *new*(*StructureDefinitionPtr*, *Init*(*gPatternPos*, *gStructPrefixes*, *gConstructorNr*, *gParams*, *gStructFields*));
end;

1271. 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 *gClusterSort* is populated when the parser finishes a cluster registration when invoking *extItemObj.FinishAntecedent* (§1281) or similar methods.

The *gClusterTerm* is populated in the *extItemObj.FinishClusterTerm* method (§1282).

⟨ Local variables for parser additions 1209 ⟩ +≡
gAntecedent, *gConsequent*: *PList*;
gClusterTerm: *TermPtr*;

1272. ⟨ Pop a registration item 1272 ⟩ ≡
itCluster: **begin case** *gClusterSort* **of**
  ExistentialRegistration: **begin**
      nLastWSItem↑.*nContent* ← *new*(*EClusterPtr*, *Init*(*nItemPos*, *gConsequent*, *gLastType*));
    include(*gCorrectnessConditions*, *syExistence*)
    end;
  ConditionalRegistration: **begin** *nLastWSItem*↑.*nContent* ← *new*(*CClusterPtr*, *Init*(*nItemPos*,
      gAntecedent, *gConsequent*, *gLastType*)); *include*(*gCorrectnessConditions*, *syCoherence*);
    end;
  FunctorialRegistration: **begin** *nLastWSItem*↑.*nContent* ← *new*(*FClusterPtr*, *Init*(*nItemPos*,
      gClusterTerm, *gConsequent*, *gLastType*)); *include*(*gCorrectnessConditions*, *syCoherence*);
    end;
  endcases;
  end;

See also section 1275.

This code is used in section 1250.

1273. Pop a registration item. For an `identify` or `reduce` registration, we assign the content of the caller’s *nLastWSItem* a new *IdentifyRegistration* (resp., *ReduceRegistration*) object. Identify registrations use the *gIdentifyEqLociList* local variable, while the reduction registrations use the *gLeftTermInReduction* module-wide variable.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gLeftTermInReduction: *TermPtr*;

1274. ⟨ Local variables for parser additions 1209 ⟩ +≡
gIdentifyEqLociList: *PList*;

1275. ⟨ Pop a registration item 1272 ⟩ +≡
itIdentify: **begin** *nLastWSItem*↑.*nContent* ← *new*(*IdentifyRegistrationPtr*, *Init*(*nItemPos*, *gNewPattern*,
    gPattern, *gIdentifyEqLociList*)); *include*(*gCorrectnessConditions*, *syCompatibility*);
  end;
itReduction: **begin** *nLastWSItem*↑.*nContent* ← *new*(*ReduceRegistrationPtr*, *Init*(*nItemPos*,
    gLeftTermInReduction, *gLastTerm*)); *include*(*gCorrectnessConditions*, *syReducibility*);
  end;
itPropertyRegistration: *SethoodRegistrationPtr*(*nLastWSItem*↑.*nContent*)↑.*nJustification* ← *nInference*;

1276. 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* ← *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. . . ⟧

⟨ Check the popped item's linkages are valid 1276 ⟩ ≡
  with *extBlockPtr*(*gBlockPtr*)↑ **do**
    begin if *nLinked* **then**
      begin *Error*(*nLinkPos*, 178); *nLinked* ← *false* **end**;
    nLinkAllowed ← *nLinkable*; *nLinkProhibited* ← ¬*nLinkable*;
    if ¬*StillCorrect* **then**
      begin *nLinkAllowed* ← *false*; *nLinkProhibited* ← *false* **end**;
    end

This code is used in section 1249.

Subsection 23.2.3. Registrations and notations

1277. Processing synonyms. We need to update the *gNewPatternPos* and *gNewPattern* global variables when processing a synonym.

  define *process_notation_item* ≡ *gNewPatternPos* ← *gPatternPos*; *gNewPattern* ← *gPattern*

⟨ Extended item implementation 1226 ⟩ +≡
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**;

1278. Starting attributes. This is used when the parser encounters a cluster registration (§1643). The *gAttrColl* is populated in the *extSubexpObj.CompleteAdjectiveCluster* (§1398) method.

⟨ Local variables for parser additions 1209 ⟩ +≡
gAttrColl: *PList*;

1279. ⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartAttributes*;
  begin *gAttrColl* ← *new*(*PList*, *Init*(6));
  end;

1280. Starting a sentence. We just need to populate the caller's *nPropPos*, assigning to it the current position of the parser.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartSentence*;
  begin *nPropPos* ← *CurPos*;
  end;

1281. Processing conditional registration. This populates the *gClusterSort* and the related global variables, as the parser finishes parsing the antecedent and consequent to the cluster.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishAntecedent*;
  begin *gClusterSort* ← *ConditionalRegistration*; *gAntecedent* ← *gAttrColl*;
  end;

procedure *extItemObj.FinishConsequent*;
  begin *gConsequent* ← *gAttrColl*;
  end;

1282. Finishing a cluster. This populates the *gClusterSort* and the *gClusterTerm*.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishClusterTerm*;
  begin *gClusterSort* ← *FunctorialRegistration*; *gClusterTerm* ← *gLastTerm*;
  end;

1283. Identify registration. Schematically, we have the registration statement look like (using global variable names for the subexpressions):

`identify` ⟨*gNewPattern*⟩ `with` ⟨*gPattern*⟩ [`when` ⟨*gIdentifyEqLociList*⟩];

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.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartFuncIdentify*;
  begin end;

procedure *extItemObj.ProcessFuncIdentify*;
  begin *gNewPatternPos* ← *gPatternPos*; *gNewPattern* ← *gPattern*;
  end;

procedure *extItemObj.CompleteFuncIdentify*;
  begin *gIdentifyEqLociList* ← **nil**;
  if *CurWord.Kind* = *sy_When* **then** *gIdentifyEqLociList* ← *new*(*PList*, *Init*(0));
  end;

1284. "Reduces to" registrations. Recall, these schematically look like

`reduce` ⟨*gLeftLocus*⟩ `to` ⟨*Locus*⟩;

Mizar will populate *gLeftLocus*. The gambit will be to treat this as a functor pattern; i.e., the *gLeftLocus* will be used to populate *gNewPattern* in the method *extItemObj.FinishFunctorPattern* (§1323).

⟨ Local variables for parser additions 1209 ⟩ +≡
gLeftLocus: *LocusPtr*;

1285. ⟨ Extended item implementation 1226 ⟩ +≡
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

1286. The terminology used by the parser appears to be (§§1565 *et seq.*):

`let` ⟨*Fixed Variables*⟩`;`

and

`consider` ⟨*Fixed Variables*⟩ `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

⟨*Fixed-Variables*⟩ ::= ⟨*Implicitly-Qualified-Variables*⟩ { `","` ⟨*Fixed-Variables*⟩ }
    | ⟨*Explicitly-Qualified-Variables*⟩ { `","` ⟨*Fixed-Variables*⟩ }
⟨*Implicitly-Qualified-Variables*⟩ ::= ⟨*Variables*⟩
⟨*Explicitly-Qualified-Variables*⟩ ::= ⟨*Qualified-Segment*⟩ { `","` ⟨*Qualified-Segment*⟩ }
⟨*Qualified-Segment*⟩ ::= ⟨*Variables*⟩ ⟨*Qualification*⟩
⟨*Variables*⟩ ::= ⟨*Variable*⟩ { `","` ⟨*Variable*⟩ }
⟨*Qualification*⟩ ::= (`"be"` | `"being"`) ⟨*Type*⟩

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:

⟨*Loci-Declaration*⟩ ::= `"let"` ⟨*Qualified-Variables*⟩[`"such"` ⟨*Conditions*⟩] `;`

The grammar for a qualified segment *requires* implicitly qualified variables appear at the very end.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartFixedVariables*;
  begin *gQualifiedSegmentList* ← *new*(*PList*, *Init*(0));
  end;

1287. ⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gQualifiedSegment: *MList*;
gSegmentPos: *Position*;

1288. Fixed segments. This refers to each "explicitly qualified segment" or "implicitly qualified segment" appearing in the fixed variables portion. The fixed segments are separated by commas.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartFixedSegment*;
  begin *gQualifiedSegment.Init*(0); *gSegmentPos* ← *CurPos*;
  end;

1289. When parsing fixed variables, and the parser has just entered the loop to parse fixed variables, this function will be invoked.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.ProcessFixedVariable*;
  begin *gQualifiedSegment.Insert*(*new*(*VariablePtr*, *Init*(*CurPos*, *GetIdentifier*)));
  end;

1290. This "clears the cache" for assigning the type in an explicitly qualified segment (appearing in a fixed variable segment).

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.ProcessBeing*;
  begin *gLastType* ← **nil**;
  end;

1291. The last statement in the parser loop when parsing "fixed variables" is to push the "fixed segment" onto the *gQualifiedSegmentList* global variable. There are two cases to consider: the implicitly qualified variables and the explicitly qualified variables.

The implicitly qualified case simple *moves* the pointers around "manually", so we need to update every entry of *gQualifiedSegment.Items* to be nil. The explicitly qualified case moves the pointers around using the *MList* constructor, mutating *gQualifiedSegment* into a list of **nil** pointers.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishFixedSegment*;
  var *k*: *integer*;
  begin if *gLastType* ≠ **nil then** { explicitly qualified case }
    begin *gQualifiedSegmentList*↑.*Insert*(*new*(*ExplicitlyQualifiedSegmentPtr*, *Init*(*gSegmentPos*,
        new(*PList*, *MoveList*(*gQualifiedSegment*)), *gLastType*))); *gQualifiedSegment.DeleteAll*;
    end
  else begin for *k* ← 0 **to** *gQualifiedSegment.Count* − 1 **do**
      begin *gQualifiedSegmentList*↑.*Insert*(*new*(*ImplicitlyQualifiedSegmentPtr*,
          Init(*VariablePtr*(*gQualifiedSegment.Items*↑[*k*])↑.*nVarPos*, *gQualifiedSegment.Items*↑[*k*])));
      gQualifiedSegment.Items↑[*k*] ← **nil**;
      end;
    end;
  gQualifiedSegment.Done;
  end;

1292. When we finish parsing fixed variables, we need to "unset" the *gPremises* global variable. The parser will either be looking at a semicolon token or at "`such` ⟨*Conditions*⟩". The reader should note that *gSuchThatOcc* is not used in the parser, nor anywhere else in Mizar. But we recall (§1262) the *gSuchPos* is used when popping a `let` statement.

⟨ Local variables for parser additions 1209 ⟩ +≡
gSuchThatOcc: *boolean*; { not used }

1293. ⟨ Extended item implementation 1226 ⟩ +≡

procedure *extItemObj.FinishFixedVariables*;
  begin *gSuchThatOcc* ← *CurWord.Kind* = *sy_Such*; *gSuchPos* ← *CurPos*; *gPremises* ← **nil**;
  end;

1294. When the parser encounters the statement:

`let` ⟨*Fixed-Variables*⟩ `such that` ⟨*Assumption*⟩;

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.

⟨ Extended item implementation 1226 ⟩ +≡

procedure *extItemObj.StartAssumption*;
  begin *gPremises* ← *new*(*PList*, *Init*(0)); *gThatPos* ← *CurPos*;
  end;

1295. Finishing an assumption will update the global variable *gBlockPtr*'s field reflecting it has assumptions.

⟨ Extended item implementation 1226 ⟩ +≡

procedure *extItemObj.FinishAssumption*;
  begin *ExtBlockPtr*(*gBlockPtr*)↑.*nHasAssumptions* ← *true*;
  end;

1296. When the Mizar parser has encountered

`assume that` ⟨*Conditions*⟩;

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.

⟨ Extended item implementation 1226 ⟩ +≡

procedure *extItemObj.StartCollectiveAssumption*;
  begin *gPremises* ← *new*(*PList*, *Init*(0)); *gThatPos* ← *CurPos*;
  end;

1297. Processing copula in a definition. When defining a (nonexpandable) mode, a functor, a predicate, or an attribute, we have

$$\langle \mathit{Pattern} \rangle\ \texttt{means}\ \langle \mathit{Expression} \rangle;$$

or

$$\langle \mathit{Pattern} \rangle\ \texttt{equals}\ \langle \mathit{Expression} \rangle;$$

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);
- 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.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gDefLabId: *integer*;
gDefLabPos: *Position*;

1298. ⟨ Local variables for parser additions 1209 ⟩ +≡
gOtherwise: *PObject*;

1299. ⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.ProcessMeans*;
  begin *gDefLabId* ← 0; *gDefLabPos* ← *CurPos*; *gDefiningWay* ← *dfMeans*; *gOtherwise* ← **nil**;
  gMeansPos ← *CurPos*
  end;

procedure *extItemObj.ProcessEquals*;
  begin *gDefLabId* ← 0; *gDefLabPos* ← *CurPos*; *gDefiningWay* ← *dfEquals*; *gOtherwise* ← **nil**;
  gMeansPos ← *CurPos*;
  end;

1300. When parsing a definition-by-cases, the cases are terminated with an "`otherwise`" keyword. Recall the grammar for such definitions looks like:

$$\langle \mathit{Partial\text{-}Definiens\text{-}List} \rangle\ \texttt{"otherwise"}\ \langle \mathit{Expression} \rangle;$$

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*.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishOtherwise*;
  begin if *gDefiningWay* = *dfEquals* **then** *gOtherwise* ← *gLastTerm*
  else *gOtherwise* ← *gLastFormula*;
  end;

1301. Starting a definiens should mutate the *it_Allowed* global variable to be equal to the caller's *nItAllowed* field. The *it_Allowed* global variable is toggled on and off when the parser encounters "guards" in conditional definitions, whereas the *nItAllowed* fields reflects whether the sort of definition allows "`it`" in the definiens.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartDefiniens*;
  begin *it_Allowed* ← *nItAllowed*;
  end;

1302. "Guards" refers to the conditions in a definition-by-cases. Specifically, we have

⟨*Partial-Definiens*⟩ ::= ⟨*Expression*⟩ `"if"` ⟨*Guard-Formula*⟩

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.

⟨ Local variables for parser additions 1209 ⟩ +≡
gPartDef: *PObject*;

1303. ⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartGuard*;
  begin if *gPartialDefs* = **nil then** *gPartialDefs* ← *new*(*PList*, *Init*(0));
  it_Allowed ← *false*;
  if *gDefiningWay* = *dfMeans* **then** *gPartDef* ← *gLastFormula*
  else *gPartDef* ← *gLastTerm*;
  end;

1304. After parsing a formula, then the parser will invoke *FinishGuard*. This will append to *gPartialDefs* a new partial definiens.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishGuard*;
  begin *it_Allowed* ← *nItAllowed*;
  case *gDefiningWay* **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*)));
  endcases;
  end;

1305. Recall for functor definitions we have something like:

$$\texttt{func}\ \langle \textit{Pattern} \rangle\ \texttt{->}\ \langle \textit{Type} \rangle\ (\ \texttt{means}\ |\ \texttt{equals}\ )\ \ldots$$

Similarly, nonexpandable modes look like

$$\texttt{mode}\ \langle \textit{Pattern} \rangle\ \texttt{->}\ \langle \textit{Type} \rangle\ \texttt{means}\ \ldots$$

The "`->` ⟨*Type*⟩" 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.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishSpecification*;
  begin *gSpecification* ← *gLastType*;
  end;

1306. "Construction type" is the term used by the parser for "nonexpandable modes". They, too, have a type specification. The *FinishConstructionType* populates the *gSpecification* global variable with this type.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishConstructionType*;
  begin *gSpecification* ← *gLastType*;
  end;

1307. Expandable mode definitions, after encountering the "`is`" keyword, invokes the *StartExpansion* method. This just ensures there is no definiens, and the *gExpandable* global variable is assigned to "true".

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartExpansion*;
  begin if *gRedefinitions* **then** *ErrImm*(271);
  nDefiniensProhibited ← *true*; *gExpandable* ← *true*;
  end;

1308. The parser, when determining the pattern for an attribute (§1608), resets the state when starting to determine the pattern for the attribute. This is handled by the *StartAttribute* method.

We should remind the reader that attributes can only have arguments *to its left*.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gParamNbr: *integer*;

1309. ⟨ Local variables for parser additions 1209 ⟩ +≡
gLocus: *LocusPtr*;

1310. ⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartAttributePattern*;
  begin *gParamNbr* ← 0; *gParams* ← **nil**; *gLocus* ← *new*(*LocusPtr*, *Init*(*CurPos*, *GetIdentifier*));
  end;

1311. Since an attribute can only have attributes to its left, it's pretty clear when the attribute pattern has been parsed: the parser has found the attribute being defined. In that case (assuming we're not panicking), we should add the attribute format to the *gFormatsColl* dictionary and update the global variables.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishAttributePattern*;
  var *lFormatNr*: *integer*;
  begin *lFormatNr* ← 0;
  if (*CurWord.Kind* = *AttributeSymbol*) ∧ *stillcorrect* **then**
    lFormatNr ← *gFormatsColl.CollectPrefixForm*(´V´, *CurWord.Nr*, *gParamNbr*);
  gPatternPos ← *CurPos*; *gConstructorNr* ← *CurWord.Nr*;
  gPattern ← *new*(*AttributePatternPtr*, *Init*(*gPatternPos*, *gLocus*, *gConstructorNr*, *gParams*));
  end;

1312. A mode definition may include a "`sethood`" property. This particular function is used when registering sethood in a registration block.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishSethoodProperties*;
  begin
    nLastWSItem↑.*nContent* ← *new*(*SethoodRegistrationPtr*, *Init*(*nItemPos*, *gPropertySort*, *gLastType*));
  end;

1313. We remind the reader the grammar for a mode pattern

$$\langle \textit{Mode-Pattern} \rangle ::= \langle \textit{Mode-Symbol} \rangle \text{ [} \texttt{"of"} \ \langle \textit{Loci} \rangle \text{]}$$

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.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartModePattern*;
  begin *gParamNbr* ← 0; *gParams* ← **nil**; *gPatternPos* ← *CurPos*; *gConstructorNr* ← *CurWord.Nr*;
  end;

1314. Finishing a mode pattern should build a new *ModePatternObj*, and store it in the *gPattern* global variable. And if we are not panicking, we should add it to the *gFormatsColl* dictionary.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishModePattern*;
  var *lFormatNr*: *integer*;
  begin *lFormatNr* ← 0;
  if *StillCorrect* **then** *lFormatNr* ← *gFormatsColl.CollectPrefixForm*(´M´, *gConstructorNr*, *gParamNbr*);
  gPattern ← *new*(*ModePatternPtr*, *Init*(*gPatternPos*, *gConstructorNr*, *gParams*));
  end;

1315. When parser starts parsing a new predicate pattern, we should reset the relevant global variables.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartPredicatePattern*;
  begin *gParamNbr* ← 0; *gParams* ← **nil**;
  end;

1316. When the parser tries to parse a "predicative formula" (i.e., a formula involving a predicate) — including predicate patterns — the first thing it does is invoke this *ProcessPredicateSymbol* method. This resets the global variables needed to populate the arguments to the predicate in the formula.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gLeftLociNbr: *integer*;

1317. ⟨ Local variables for parser additions 1209 ⟩ +≡
gLeftLoci: *PList*;

1318. ⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.ProcessPredicateSymbol*;
  begin *gPatternPos* ← *CurPos*; *gLeftLociNbr* ← *gParamNbr*; *gLeftLoci* ← *gParams*; *gParamNbr* ← 0;
  gParams ← **nil**; *gConstructorNr* ← *CurWord.Nr*;
  end;

1319. Finishing a predicate pattern will create a new *PredicatePattern* object, update the *gPattern* global variable to point to it, and (if the parser is not panicking) add the predicate's format to the *gFormatsColl* dictionary.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishPredicatePattern*;
  var *lFormatNr*: *integer*;
  begin *lFormatNr* ← 0;
  if *StillCorrect* **then**
    lFormatNr ← *gFormatsColl.CollectPredForm*(*gConstructorNr*, *gLeftLociNbr*, *gParamNbr*);
  gPattern ← *new*(*PredicatePatternPtr*, *Init*(*gPatternPos*, *gLeftLoci*, *gConstructorNr*, *gParams*));
  end;

1320. Functor patterns a bit trickier. When starting one, what should occur depends on the type of functor being defined. Specifically, we handle brackets differently than other functors, and within the brackets we handle braces (i.e., definitions like $\{x_1, \ldots, x_n\}$) differently than square brackets ($[x_1, \ldots, x_n]$) differently than everything other functor bracket.

In all cases, even non-bracket functors, we need to reset the *gParamNbr* and *gParams* global variables so they may be populated correctly.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gSubItemKind: *TokenKind*;

1321. ⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartFunctorPattern*;
  begin *gPatternPos* ← *CurPos*; *gSubItemKind* ← *CurWord.Kind*;
  case *CurWord.Kind* **of**
  LeftCircumfixSymbol: *gConstructorNr* ← *CurWord.Nr*;
  sy_LeftSquareBracket: **begin** *gSubItemKind* ← *LeftCircumfixSymbol*; *gConstructorNr* ← *SquareBracket*
    end;
  sy_LeftCurlyBracket: **begin** *gSubItemKind* ← *LeftCircumfixSymbol*; *gConstructorNr* ← *CurlyBracket*
    end;
  othercases *gConstructorNr* ← 0;
  endcases; *gParamNbr* ← 0; *gParams* ← **nil**;
  end;

1322. For "non-bracket" functors (i.e., infix operators), the functor pattern is processed by (1) getting the left parameters, (2) processing the functor symbol, (3) getting the right parameters. This function is precisely step (2).

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.ProcessFunctorSymbol*;
  begin *gPatternPos* ← *CurPos*;
  if *CurWord.Kind* = *InfixOperatorSymbol* **then**
    begin *gSubItemKind* ← *InfixOperatorSymbol*; *gConstructorNr* ← *CurWord.Nr*;
    gLeftLociNbr ← *gParamNbr*; *gLeftLoci* ← *gParams*; *gParamNbr* ← 0; *gParams* ← **nil**;
    end;
  end;

1323. When defining a bracket functor pattern, we add a new bracket format to the *gFormatsColl* dictionary, and then set *gPattern* to a newly allocated Bracket pattern.

When defining an infix functor, we add a new functor format to the *gFormatsColl* dictionary, and then we set the *gPattern* to a newly allocated infix functor pattern.

The "other cases" constructs an infix functor pattern, but does not add the form to the *gFormatsColl* dictionary.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishFunctorPattern*;
  var *lConstructorNr*, *lFormatNr*: *integer*;
  begin *lFormatNr* ← 0;
  case *gSubItemKind* **of**
  LeftCircumfixSymbol: **begin** *lConstructorNr* ← *CurWord.Nr*;
    if *StillCorrect* **then**
      lFormatNr ← *gFormatsColl.CollectBracketForm*(*gConstructorNr*, *lConstructorNr*, *gParamNbr*, 0, 0);
    gPattern ← *new*(*CircumfixFunctorPatternPtr*, *Init*(*gPatternPos*, *gConstructorNr*, *lConstructorNr*,
        gParams));
    end;
  InfixOperatorSymbol: **begin if** *StillCorrect* **then**
      lFormatNr ← *gFormatsColl.CollectFuncForm*(*gConstructorNr*, *gLeftLociNbr*, *gParamNbr*);
    gPattern ← *new*(*InfixFunctorPatternPtr*, *Init*(*gPatternPos*, *gLeftLoci*, *gConstructorNr*, *gParams*));
    end;
  othercases
      gPattern ← *new*(*InfixFunctorPatternPtr*, *Init*(*gPatternPos*, *gLeftLoci*, *gConstructorNr*, *gParams*));
  endcases;
  end;

1324. The Parser's *ReadVisible* procedure begins by invoking this *StartVisible* method. The *ReadVisible* procedure occurs when getting most patterns.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartVisible*;
  begin *gParams* ← *new*(*PList*, *Init*(0));
  end;

1325. The Parser iteratively calls its *GetVisible* (§1602) procedure when *ReadVisible* arguments in a pattern. The *GetVisible* procedure in turn invokes this *ProcessVisible*, which increments the number of parameters, and pushes a new *Locus* object onto the *gParams* stack.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.ProcessVisible*;
  begin *inc*(*gParamNbr*);
  if *gParams* ≠ **nil then** *gParams*↑.*Insert*(*new*(*LocusPtr*, *Init*(*CurPos*, *GetIdentifier*)));
  end;

1326. Recall a structure definition, when it has ancestors, looks like

`struct` (⟨*Ancestors*⟩) ⟨*Structure-Symbol*⟩ ⋯

The ⟨*Ancestors*⟩ field is considered the "prefix" to the structure definition. The Parser parses a type (thereby populating the *gLastType* global variable), then invokes the *FinishPrefix* method, then iterates if it encounters a comma.

The *FinishPrefix* method pushes the *gLastType* global variable to the *gStructPrefixes* state variable.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishPrefix*;
  begin *gStructPrefixes*.*Insert*(*gLastType*);
  end;

1327. ⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.ProcessStructureSymbol*;
  var *lFormatNr*: *integer*;
  begin *gConstructorNr* ← 0; *gPatternPos* ← *CurPos*;
  if *CurWord.Kind* = *StructureSymbol* **then** *gConstructorNr* ← *CurWord.Nr*;
  lFormatNr ← *gFormatsColl.CollectPrefixForm*(´J´, *gConstructorNr*, 1); *gParamNbr* ← 0;
  gParams ← **nil**;
  end;

1328. When the Parser has just finished parsing the ancestors to a structure, but has not parsed the visible arguments. Then the Parser prepares for reading the visible arguments and then the fields by invoking this method. This initializes the *gStructFields* state variable as well as the *gFieldsNbr* state variable.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gFieldsNbr: *integer*;

1329. ⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartFields*;
  var *lFormatNr*: *integer*;
  begin *lFormatNr* ← *gFormatsColl.CollectPrefixForm*(´L´, *gConstructorNr*, *gParamNbr*);
  in_AggrPattern ← *true*; *gStructFields* ← *new*(*PList*, *Init*(0)); *gFieldsNbr* ← 0;
  end;

1330. The Parser has just encountered the end structure bracket ("`#)`") token, so we want to add the format to the *gFormatsColl* dictionary.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishFields*;
  var *lFormatNr*: *integer*;
  begin *lFormatNr* ← *gFormatsColl.CollectPrefixForm*(´G´, *gConstructorNr*, *gFieldsNbr*);
  end;

1331. Recall that each field-segment looks like

⟨*Field-Segment*⟩ ::= ⟨*Selector-Symbol*⟩ {"," ⟨*Selector-Symbol*⟩} ⟨*Specification*⟩

Before parsing the field-segment, the *StartAggrPattSegment* is invoked.

⟨ Local variables for parser additions 1209 ⟩ +≡
gStructFieldsSegment: *PList*;
gSgmPos: *Position*;

1332. ⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartAggrPattSegment*;
  begin *gStructFieldsSegment* ← *new*(*Plist*, *Init*(0)); *gSgmPos* ← *CurPos*;
  end;

1333. For each selector-symbol the Parser encounters, it invokes the *ProcessField*.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.ProcessField*;
  var *lFormatNr*: *integer*;
  begin *lFormatNr* ← *gFormatsColl.CollectPrefixForm*(´U´, *CurWord.Nr*, 1);
  gStructFieldsSegment↑.*Insert*(*new*(*FieldSymbolPtr*, *Init*(*CurPos*, *CurWord.Nr*))); *inc*(*gFieldsNbr*);
  end;

1334. After each field has been parsed, the Parser invokes this method to update the *gStructFields* will push a new field segment object onto it.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishAggrPattSegment*;
  begin *gStructFields.Insert*(*new*(*FieldSegmentPtr*, *Init*(*gSgmPos*, *gStructFieldsSegment*, *gLastType*)));
  end;

Subsection 23.2.5. Processing remaining statements

1335. Processing schemes. Most of these methods are used in parsing a scheme block (§1654). It will be useful to examine that function to see where these methods are invoked.

When the Parser starts a new scheme, several state variables need to be reset. The *gSchemeIdNr* is populated by the *GetIdentifier* (§1201) procedure, the *gSchemeIdPos* is assigned the current position, and the *gSchemeParams* should be allocated to an empty list.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.ProcessSchemeName*;
  begin *gSchemeIdNr* ← *GetIdentifier*; *gSchemeIdPos* ← *CurPos*;
  gSchemeParams ← *new*(*PList*, *Init*(0));
  end;

1336. A scheme qualification segment looks like, for predicates:

$$\langle \textit{Variable} \rangle \; \{ \;\; \texttt{","} \; \langle \textit{Variable} \rangle \; \} \; \texttt{"["} [\langle \textit{Type-Expression-List} \rangle] \texttt{"]"}$$

And for functors:

$$\langle \textit{Variable} \rangle \; \{ \;\; \texttt{","} \; \langle \textit{Variable} \rangle \; \} \;\; \texttt{"("} \; [\langle \textit{Type-Expression-List} \rangle] \; \texttt{")"}$$

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.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gTypeList: *MList*;

1337. ⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartSchemeQualification*;
  begin *gSubItemKind* ← *CurWord.Kind*; *gTypeList.Init*(4);
  end;

1338. After the type-list has been parsed, but before the closing parentheses or bracket has been encountered, the Parser invokes the *FinishSchemeQualification* method. This assigns the current position to the *gSubItemPos*.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gSubItemPos: *Position*;

1339. ⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishSchemeQualification*;
  begin *gSubItemPos* ← *CurPos*
  end;

1340. Starting a scheme segment describes the situation where we are *just about* to start parsing the comma-separated list of identifiers for the scheme parameters. This just assigns the current position to the *gSubItemPos*, then initializes *gSchVarIds* to 2 spots.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gSchVarIds: *MList*;

1341. ⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartSchemeSegment*;
  begin *gSubItemPos* ← *CurPos*; *gSchVarIds.Init*(2);
  end;

1342. After parsing the identifier for an entry in the comma-separated list of scheme variables, the Parser invokes *ProcessSchemeVariable* to add the recently parsed identifier to the *gSchVarIds* state variable.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.ProcessSchemeVariable*;
  begin *gSchVarIds.Insert*(*new*(*VariablePtr*, *Init*(*CurPos*, *GetIdentifier*)));
  end;

1343. Once the list of scheme variables and their type specification has been parsed, then the Parser invokes the *FinishSchemeSegment* method. This just turns the *gSchVarIds* list into a Predicate segment or a Functor segment, using the type list the Parser just finished parsing.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishSchemeSegment*;
  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;

1344. The "scheme thesis" is the formula statement of the scheme. Informally, a scheme looks like:

`scheme` {⟨*Scheme-Parameters*⟩} ⟨*Scheme-thesis*⟩ `"provided"` ⟨*Scheme-premises*⟩

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*.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishSchemeThesis*;
  begin *gSchemeConclusion* ← *gLastFormula*; *gSchemePremises* ← *new*(*Plist*, *Init*(0));
  end;

1345. The premises for a scheme consists of finitely many formulas separated by "`and`" keywords. The Parser enters into a loop invoking this method *after* parsing the formula but *before* checking the next word is "`and`" (and iterating loop). We just need to push the formula onto the *gSchemePremises* list.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishSchemePremise*;
  begin *gSchemePremises*↑.*Insert*(*new*(*PropositionPtr*, *Init*(*nLabel*, *gLastFormula*, *nPropPos*)));
  end;

1346. Reserved variables. These methods are invoked only when the Parser parses a reservation (§1651). A "reservation segment" refers to the comma-separated list of variables and the type.

Starting a reservation segment allocates a new (empty) list for *gResIdents*, and assigns the *gResPos* to the current position. Each variable encountered in the comma-separated list of variables is appended to the *gResIdents* list using the *ProcessReservedIdentifier* method.

Mizar treats each reservation segment as a separate statement. So there is no difference between:

```
reserve G for Group, x,y,z for Element of G;
```

...and...

```
reserve G for Group;
reserve x,y,z for Element of G;
```

Finishing a reservation mutates both the *gLastWSItem* and *gLastWSBlock* global variables. Specifically, we allocate a new reservation *Item*, then update *gLastWSItem* to point to it. The caller's *nLastWSItem* is updated to point to it, too. We assign the content of this newly allocated reservation *Item* based on the *gResIdents* list. We insert this *Item* to the end of the *gLastWSBlock*'s items.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gResIdents: *PList*;
gResPos: *Position*;

1347. ⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartReservationSegment*;
  begin *gResIdents* ← *new*(*Plist*, *Init*(0)); *gResPos* ← *CurPos*;
  end;
procedure *extItemObj.ProcessReservedIdentifier*;
  begin *gResIdents*↑.*Insert*(*new*(*VariablePtr*, *Init*(*CurPos*, *GetIdentifier*)));
  end;
procedure *extItemObj.FinishReservationSegment*;
  begin *gLastWSItem* ← *gWsTextProper*↑.*NewItem*(*itReservation*, *gResPos*);
  nLastWSItem ← *gLastWSItem*;
  gLastWSItem↑.*nContent* ← *new*(*ReservationSegmentPtr*, *Init*(*gResIdents*, *gLastType*));
  gLastWSItem↑.*nItemEndPos* ← *PrevPos*; *gLastWSBlock*↑.*nItems*.*Insert*(*gLastWSItem*);
  end;

1348. Both "`defpred`" and "`deffunc`" invokes *StartPrivateDefiniendum* to initialize the *gTypeList*, store the identifier in the *gPrivateId*, and assign the current position to the *gPrivateIdPos*. Further, *dol_Allowed* is toggled to *true* — placeholder variables are going to be allowed in the type declarations of the private functor or private predicate (for example "`defpred Foo[set, Element of $1]`").

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gPrivateId: *Integer*;
gPrivateIdPos: *Position*;

1349. ⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartPrivateDefiniendum*;
  begin *gPrivateId* ← *GetIdentifier*; *gPrivateIdPos* ← *CurPos*; *dol_Allowed* ← *true*; *gTypeList*.*Init*(4);
  end;

1350. Reading a "type list" (for scheme parameters or for private definitions) loops over reading a type, then pushing it onto the *gTypeList*. The parser delegates that latter "push work" to the *FinishLocusType* method.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishLocusType*;
  begin *gTypeList.Insert*(*gLastType*);
  end;

1351. The life-cycle of expressions is a little convoluted. The *Item* will allocate a new *extExpression* object and assign it to the *gExpPtr*. Later, almost always, the *gExpPtr* will invoke a method to create a subexpression. This subexpression will be populated, then the *gLastTerm* (or *gLastFormula*) will be updated to point to this subexpression object. The expression object will be freed.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.CreateExpression*(*fExpKind* : *ExpKind*);
  begin *gExpPtr* ← *new*(*extExpressionPtr*, *Init*(*fExpKind*));
  end;

1352. Recall the "set" statement is of the form

"set" ⟨*Variable*⟩ "=" ⟨*Term*⟩ { "," ⟨*Variable*⟩ "=" ⟨*Term*⟩ }

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.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishPrivateConstant*;
  begin *gLastWSItem* ← *gWsTextProper*↑.*NewItem*(*itConstantDefinition*, *nItemPos*);
  nLastWSItem ← *gLastWSItem*; *gLastWSItem*↑.*nContent* ← *new*(*ConstantDefinitionPtr*,
    Init(*new*(*VariablePtr*, *Init*(*gPrivateIdPos*, *gPrivateId*)), *gLastTerm*));
  gLastWSItem↑.*nItemEndPos* ← *PrevPos*; *gLastWSBlock*↑.*nItems*.*Insert*(*gLastWSItem*);
  nItemPos ← *CurPos*;
  end;

1353. When the Parser is about to start parsing an assignment "⟨*Variable*⟩ = ⟨*Term*⟩" in a "set" statement, the Parser invokes this method. The caller assigns the *gPrivateId* state variable to be the result of *GetIdentifier*, and the *gPrivateIdPos* state variable to be the current position.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartPrivateConstant*;
  begin *gPrivateId* ← *GetIdentifier*; *gPrivateIdPos* ← *CurPos*;
  end;

1354. For a "defpred" and a "deffunc", before parsing the definiens, we need to set the *dol_Allowed* global variable to true (to allow placeholder variables).

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartPrivateDefiniens*;
  begin *dol_Allowed* ← *true*;
  end;

1355. 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.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishPrivateFuncDefinienition*;
  begin *nLastWSItem*↑.*nContent* ← *new*(*PrivateFunctorDefinitionPtr*, *Init*(*new*(*VariablePtr*, *Init*(*gPrivateIdPos*, *gPrivateId*)), *new*(*PList*, *MoveList*(*gTypeList*)), *gLastTerm*));
  end;

1356. When finishing the definiendum formula for a "`defpred`", the Parser invokes this *FinishPrivatePredDefinienition* method.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishPrivatePredDefinienition*;
  begin *nLastWSItem*↑.*nContent* ← *new*(*PrivatePredicateDefinitionPtr*, *Init*(*new*(*VariablePtr*, *Init*(*gPrivateIdPos*, *gPrivateId*)), *new*(*PList*, *MoveList*(*gTypeList*)), *gLastFormula*));
  end;

1357. Reconsider statements.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.ProcessReconsideredVariable*;
  begin *gPrivateId* ← *GetIdentifier*; *gPrivateIdPos* ← *CurPos*;
  end;

procedure *extItemObj.FinishReconsideredTerm*;
  begin *gReconsiderList*↑.*Insert*(*new*(*TypeChangePtr*, *Init*(*Equating*, *new*(*VariablePtr*, *Init*(*gPrivateIdPos*, *gPrivateId*)), *gLastTerm*)));
  end;

1358. This is invoked when parsing a private item which is a "`reconsider`" statement.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishDefaultTerm*;
  begin *gReconsiderList*↑.*Insert*(*new*(*TypeChangePtr*, *Init*(*VariableIdentifier*, *new*(*VariablePtr*, *Init*(*gPrivateIdPos*, *gPrivateId*)), **nil**)));
  end;

1359. When the Parser finishes parsing a formula in "`consider` ⟨*Segment*⟩ `such that` ⟨*Formula*⟩ {`and` ⟨*Formula*⟩}", the Parser invokes the *FinishCondition* method. This checks that *gPremises* has been allocated, then pushes a new labeled formula into it.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishCondition*;
  begin if *gPremises* = **nil then** *gPremises* ← *new*(*PList*, *Init*(0));
  gPremises↑.*Insert*(*new*(*PropositionPtr*, *Init*(*nLabel*, *gLastFormula*, *nPropPos*)));
  end;

1360. In statements of the form

`assume` ⟨*Formula*⟩;

Or of the form

`assume` ⟨*Formula*⟩ `and` ⟨*Formula*⟩ `and` ... `and` ⟨*Formula*⟩;

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**.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishHypothesis*;
  begin if *gPremises* ≠ **nil then**
    gPremises↑.*Insert*(*new*(*PropositionPtr*, *Init*(*nLabel*, *gLastFormula*, *nPropPos*)));
  end;

1361. "Take" statements. For statements of the form

`take` ⟨*Variable*⟩ = ⟨*Term*⟩;

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.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.ProcessExemplifyingVariable*;
  begin *gPrivateId* ← *GetIdentifier*; *gPrivateIdPos* ← *CurPos*;
  end;

procedure *extItemObj.FinishExemplifyingVariable*;
  begin *gLastWSItem* ← *gWsTextProper*↑.*NewItem*(*itExemplification*, *nItemPos*);
  nLastWSItem ← *gLastWSItem*; *gLastWSItem*↑.*nContent* ← *new*(*ExamplePtr*, *Init*(*new*(*VariablePtr*, *Init*(*gPrivateIdPos*, *gPrivateId*)), *gLastTerm*)); *gLastWSItem*↑.*nItemEndPos* ← *PrevPos*;
  gLastWSBlock↑.*nItems*.*Insert*(*gLastWSItem*); *nItemPos* ← *CurPos*;
  end;

1362. In statements of the form

`take` ⟨*Term*⟩;

the Parser begins by invoking *StartExemplifyingTerm*, parses the term, then *FinishExemplifyingTerm*.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartExemplifyingTerm*;
  begin if $(CurWord.Kind = Identifier) \land extBlockPtr(gBlockPtr)\uparrow.nInDiffuse \land ((AheadWord.Kind = sy\_Comma) \lor (AheadWord.Kind = sy\_Semicolon))$ **then**
    begin $gPrivateId \leftarrow GetIdentifier$; $gPrivateIdPos \leftarrow CurPos$;
    end
  else $gPrivateId \leftarrow 0$;
  end;

procedure *extItemObj.FinishExemplifyingTerm*;
  begin $gLastWSItem \leftarrow gWsTextProper\uparrow.NewItem(itExemplification, nItemPos)$;
  $nLastWSItem \leftarrow gLastWSItem$;
  if $gPrivateId \neq 0$ **then** $gLastWSItem\uparrow.nContent \leftarrow new(ExamplePtr, Init(new(VariablePtr, Init(gPrivateIdPos, gPrivateId)), \mathbf{nil}))$
  else $gLastWSItem\uparrow.nContent \leftarrow new(ExamplePtr, Init(\mathbf{nil}, gLastTerm))$;
  $gLastWSItem\uparrow.nItemEndPos \leftarrow PrevPos$; $gLastWSBlock\uparrow.nItems.Insert(gLastWSItem)$;
  $nItemPos \leftarrow CurPos$;
  end;

1363. When the Parser examines the correctness conditions (§1622), it loops over the correctness conditions and justifications. Afterwards, it invokes the *ProcessCorrectness* method, which tests that the parser is not current looking at a correctness keyword. Then it tests if *gCorrectnessConditions* is empty or *AxiomsAllowed* (in which case, correctness has been satisfies, so the Parser moves happily along). But if $gCorrectnessConditions \neq \emptyset$ or axioms are not allowed, then a 73 error is raised.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.ProcessCorrectness*;
  begin if $CurWord.Kind \neq sy\_Correctness$ **then**
    if $(gCorrectnessConditions \neq [\,]) \land \neg AxiomsAllowed$ **then** $Error(gDefPos, 73)$;
  end;

1364. A "construction type" appears in a redefinition where the type is redefined. In such a situation, we need to add "`coherence`" as a correctness condition. The *StartConstructionType* handles this task.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartConstructionType*;
  begin if $gRedefinitions \land (CurWord.Kind = sy\_Arrow)$ **then**
    $include(gCorrectnessConditions, syCoherence)$;
  end;

1365. This is used in the Parser's *ProcessLab* procedure. Really, all the work is being done here: the *nLabel* field of the caller is assigned to a newly allocated *Label* object.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.ProcessLabel*;
  begin $nLabelIdNr \leftarrow 0$; $nLabelIdPos \leftarrow CurPos$;
  if $(CurWord.Kind = Identifier) \land (AheadWord.Kind = sy\_Colon)$ **then** $nLabelIdNr \leftarrow CurWord.Nr$;
  $nLabel \leftarrow new(LabelPtr, Init(nLabelIdNr, nLabelIdPos))$;
  end;

1366. A regular statement is either a "diffuse" statement (which occurs with the "`now`" keyword) or else it's a "compact" statement.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartRegularStatement*;
  begin if *CurWord.Kind* = *sy_Now* **then** *nRegularStatementKind* ← *stDiffuseStatement*
  else *nRegularStatementKind* ← *stCompactStatement*;
  end;

1367. If the Parser encounters a colon after the copula, then it invokes this method to construct a label for the Definiens.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gDefLabel: *LabelPtr*;

1368. ⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.ProcessDefiniensLabel*;
  begin *gDefLabId* ← 0; *gDefLabPos* ← *CurPos*;
  if (*CurWord.Kind* = *Identifier*) ∧ (*AheadWord.Kind* = *sy_Colon*) **then** *gDefLabId* ← *CurWord.Nr*;
  gDefLabel ← *new*(*LabelPtr*, *Init*(*gDefLabId*, *gDefLabPos*));
  end;

1369. The Parser, having encountered "`from`" and a non-MML reference, tries to treat the identifier as the label for a scheme declared in the current article. The *nInference* field would be a *SchemeJustification* object, so we just populate its *nSchemeIdNr* and position fields.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.ProcessSchemeReference*;
  begin if *CurWord.Kind* = *Identifier* **then**
    begin *SchemeJustificationPtr*(*nInference*)↑.*nSchemeIdNr* ← *CurWord.Nr*;
    SchemeJustificationPtr(*nInference*)↑.*nSchemeInfPos* ← *CurPos*;
    end;
  end;

1370. When a "`by`" refers to a theorem or definition from an article in the MML, the Parser invokes the *StartLibraryReference* method.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gTHEFileNr: *integer*;

1371. ⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartLibraryReferences*;
  begin *gTHEFileNr* ← *CurWord.Nr*;
  end;

1372. The Parser has already encountered a "`from`" and then an MML article identifier. Before continuing to parse the scheme number, the Parser invokes this method to initialize the relevant state variables.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartSchemeLibraryReference*;
  begin *gTHEFileNr* ← *CurWord.Nr*;
  end;

1373. For references to labels found in the article being processed ("private references"), this method is invoked.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.ProcessPrivateReference*;
  begin *SimpleJustificationPtr*(*nInference*)↑.*nReferences*↑.*Insert*(*new*(*LocalReferencePtr*,
    Init(*GetIdentifier*, *CurPos*)));
  end;

1374. When using a definition from an MML article in a scheme reference (something like "`from MyScheme(ARTICLE:def 5,...)`"), well, the Parser stores this fact in a state variable *gDefinitional*. The *ProcessDef* method populates this state variable correctly.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gDefinitional: *boolean*;

1375. ⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.ProcessDef*;
  begin $gDefinitional \leftarrow (CurWord.Kind = ReferenceSort) \land (CurWord.Nr = ord(syDef))$
  end;

1376. When accumulating the references in a Scheme-Justification, and a reference is from an MML article, *ProcessTheoremNumber* transforms it into a newly allocated reference object. The caller's *nInference* then adds the newly allocated object to its *nReferences* collection.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.ProcessTheoremNumber*;
  var *lRefPtr*: *ReferencePtr*;
  begin if $CurWord.Kind \neq Numeral$ **then** *exit*;
  if $CurWord.Nr = 0$ **then**
    begin *ErrImm*(146); *exit*
    end;
  if *gDefinitional* **then** $lRefPtr \leftarrow new(DefinitionReferencePtr, Init(gTHEFileNr, CurWord.Nr, CurPos))$
  else $lRefPtr \leftarrow new(TheoremReferencePtr, Init(gTHEFileNr, CurWord.Nr, CurPos))$;
  SimpleJustificationPtr(*nInference*)↑.*nReferences*↑.*Insert*(*lRefPtr*);
  end;

1377. 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.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.ProcessSchemeNumber*;
  begin if $CurWord.Kind \neq Numeral$ **then** *exit*;
  if $CurWord.Nr = 0$ **then**
    begin *ErrImm*(146); *exit*
    end;
  with *SchemeJustificationPtr*(*nInference*)↑ **do**
    begin $nSchFileNr \leftarrow gTHEFileNr$; $nSchemeIdNr \leftarrow CurWord.Nr$; $nSchemeInfPos \leftarrow PrevPos$;
    end;
  end;

1378. This appears when the Parser starts its *Justification* (§1578) procedure, or in the *RegularStatement* (§1599) 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.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartJustification*;
  begin *nInference* ← **nil**;
  if *CurWord.Kind* = *sy_Proof* **then**
    begin if *ProofPragma* **then** *nInference* ← *new*(*JustificationPtr*, *Init*(*infProof*, *CurPos*))
    else *nInference* ← *new*(*JustificationPtr*, *Init*(*infSkippedProof*, *CurPos*))
    end;
  end;

1379. A simple justification is either a Scheme-Justification ("`from`...") , a Straightforward-Justification ("`by`..."), or... somethign else?

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartSimpleJustification*;
  begin case *CurWord.Kind* **of**
  sy_From: *nInference* ← *new*(*SchemeJustificationPtr*, *Init*(*CurPos*, 0, 0));
  sy_By: **with** *extBlockPtr*(*gBlockPtr*)↑ **do**
      nInference ← *new*(*StraightforwardJustificationPtr*, *Init*(*CurPos*, *nLinked*, *nLinkPos*));
  othercases with *extBlockPtr*(*gBlockPtr*)↑ **do**
      nInference ← *new*(*StraightforwardJustificationPtr*, *Init*(*PrevPos*, *nLinked*, *nLinkPos*));
  endcases;
  end;

1380. We should update the *nInference* field's sort to be *infError* when, well, the inference is an error (e.g., the Parser is in panic mode). We should set the *gBlockPtr*'s *nLinked* field to false when we just added a straightforward justification (or an erroneous justification).

define *is_inference_error* $\equiv \neg StillCorrect \lor ((CurWord.Kind \neq sy\_Semicolon) \land (CurWord.Kind \neq sy\_DotEquals)) \lor ((nInference\uparrow.nInfSort = infStraightforwardJustification) \land (byte(nLinked) > byte(nLinkAllowed))) \lor ((nInference\uparrow.nInfSort = infSchemeJustification) \land (SchemeJustificationPtr(nInference)\uparrow.nSchemeIdNr = 0))$

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishSimpleJustification*;
  begin with *extBlockPtr*(*gBlockPtr*)↑ **do**
    begin if *is_inference_error* **then** *nInference*↑.*nInfSort* ← *infError*;
    end;
  if (*nInference*↑.*nInfSort* = *infStraightforwardJustification*) ∨ (*nInference*↑.*nInfSort* = *infError*) **then**
    extBlockPtr(*gBlockPtr*)↑.*nLinked* ← *false*;
  end;

1381. For iterative equalities, we should recall that it looks like

```
LHS = RHS ⟨Justification⟩
    .= RHS2
    .= ...;
```

This matters because, well, when the Parser has parsed "`LHS = RHS` ⟨*Justification*⟩", 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.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishCompactStatement*;
  begin if *CurWord.Kind* = *sy_DotEquals* **then**
    begin *gIterativeLastFormula* ← *gLastFormula*; *nRegularStatementKind* ← *stIterativeEquality*;
    extBlockPtr(*gBlockPtr*)↑.*nLinked* ← *false*; *gIterativeSteps* ← *new*(*PList*, *Init*(0));
    gInference ← *nInference*;
    end;
  end;

1382. Every time the Parser encounters the "`.=`" token, it immediately invokes the *StartIterativeStep* method. This just updates the *gIterPos* state variable to the current position.

⟨ Local variables for parser additions 1209 ⟩ +≡
gIterPos: *Position*;

1383. ⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.StartIterativeStep*;
  begin *gIterPos* ← *CurPos*; **end**;

1384. Right before the Parser iterates the loop checking if "`.=`" is the next token for an iterative equation, the Parser invokes the *FinishIterativeStep* method. This just adds a new *IterativeStep* object, an AST node representing the preceding "`.= RHS by` ⟨*Justification*⟩".

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishIterativeStep*;
  begin *gIterativeSteps*↑.*Insert*(*new*(*IterativeStepPtr*, *Init*(*gIterPos*, *gLastTerm*, *nInference*)));
  end;

1385. In a definition, after the Parser finishes parsing the definiens, we construct the AST node for it with the *FinishDefiniens* method.

For each copula ("`means`" and "`equals`"), the algorithm is the same: if we just had a definition-by-cases, then store the "`otherwise`" clause in *lExp* and assign the *gDefiniens* state variable to a newly allocated conditional definiens object. If the definiens is not a definition-by-cases (i.e., it's a "simple" definition), then just assign *gDefiniens* a newly allocated *SimpleDefiniens* object.

For functor definitions (not redefinitions), the *gCorrectnessConditions* are assigned here.

⟨ Extended item implementation 1226 ⟩ +≡
procedure *extItemObj.FinishDefiniens*;
  var *lExp*: *DefExpressionPtr*;
  begin case *gDefiningWay* **of**
  dfMeans:
    if *gPartialDefs* ≠ **nil then**
      begin *lExp* ← **nil**;
      if *gOtherwise* ≠ **nil then** *lExp* ← *new*(*DefExpressionPtr*, *Init*(*exFormula*, *gOtherwise*));
      gDefiniens ← *new*(*ConditionalDefiniensPtr*, *Init*(*gMeansPos*, *gDefLabel*, *gPartialDefs*, *lExp*))
      end
    else *gDefiniens* ← *new*(*SimpleDefiniensPtr*, *Init*(*gMeansPos*, *gDefLabel*, *new*(*DefExpressionPtr*,
        Init(*exFormula*, *gLastFormula*))));
  dfEquals:
    if *gPartialDefs* ≠ **nil then**
      begin *lExp* ← **nil**;
      if *gOtherwise* ≠ **nil then** *lExp* ← *new*(*DefExpressionPtr*, *Init*(*exTerm*, *gOtherwise*));
      gDefiniens ← *new*(*ConditionalDefiniensPtr*, *Init*(*gMeansPos*, *gDefLabel*, *gPartialDefs*, *lExp*))
      end
    else *gDefiniens* ← *new*(*SimpleDefiniensPtr*, *Init*(*gMeansPos*, *gDefLabel*, *new*(*DefExpressionPtr*,
        Init(*exTerm*, *gLastTerm*))));
  endcases;
  if ¬*gRedefinitions* ∧ (*nItemKind* = *itDefFunc*) **then**
    begin if *gDefiningWay* = *dfMeans* **then** *gCorrectnessConditions* ← [*syExistence*, *syUniqueness*]
    else if *gDefiningWay* = *dfEquals* **then** *gCorrectnessConditions* ← [*syCoherence*];
    end;
  end;

Section 23.3. EXTENDED SUBEXPRESSION CLASS

1386. Aside: refactoring. We should probably refactor a private procedure *PushTermStack* to push a new term onto the term stack, and a private function *PopTermStack* to return the top of the term stack (and mutate the term stack), and possibly a *ResetTermStack* procedure (which will clear the term stack and possibly the objects stored in it?).

We see that *TermNbr* is decremented when popping the *Term* stack (via *FinishTerm*); when *FinishQualifyingFormula* is invoked, it decrements the *TermNbr*; when *FinishAttributiveFormula* is invoked, it decrements the *TermNbr*; but these latter two methods can (and should) be refactored to use the *FinishTerm* to pop the term stack and decrement the *TermNbr* state variable.

Assigning the *TermNbr* occurs when *CreateArgs* method is invoked; the *InsertIncorrBasic* method resets the *TermNbr* to the *nTermBase*; the *ProcessAtomicFormula*, when a 157 error is raised, will reset the *TermNbr* to the *nTermBase*; when the constructor for an *extExpression* object is invoked, it resets the *TermNbr* to zero (which happens in the *extItem*'s *CreateExpression* method—which occurs frequently enough to be a worry).

The only time when the *TermNbr* is incremented is when we push a new term onto the *Term* stack.

1387. There is a comment in Polish "teraz jest to kolekcja MultipleTypeExp", which Google translates to "now it is a MultipleTypeExp collection". I have made this replacement in the code below, prefixed with a "+" sign (to distinguish it from the other comment already in English).

Also note: the *nRestriction* refers to the subformula in a universally quantified formula

$$\texttt{for}\ \langle \textit{Variables} \rangle\ \texttt{st}\ \langle \textit{Restriction} \rangle\ \texttt{holds}\ \ldots$$

define *arg_type* ≡ **record** *Start*, *Length*: *integer*;
end
define *func_type* ≡ **record** *Instance*, *SymPri*: *integer*;
FuncPos: *Position*;
end

⟨ Methods implemented by subclasses of *SubexpObj* 716 ⟩ +≡

1388. ⟨ Extended subexpression class declaration 1388 ⟩ ≡

extSubexpPtr = ↑*extSubexpObj*;
extSubexpObj = **object** (*SubexpObj*)
  nTermBase, *nRightArgBase*: *integer*;
  nSubexpPos, *nNotPos*, *nRestrPos*: *Position*;
  nQuaPos: *Position*;
  nSpelling: *Integer*;
  nSymbolNr, *nRSymbolNr*: *integer*;
  nConnective, *nNextWord*: *TokenKind*;
  nModeKind: *TokenKind*;
  nModeNr: *integer*;
  nRightSideOfPredPos: *Position*;
  nMultipredicateList: *MList*;
  nSample: *TermPtr*; { for Fraenkel terms }
  nAllPos: *Position*;
  nPostQualList: *MList*; { + now it is a MultipleTypeExp collection }
  nQualifiedSegments: *MList*;
  nSegmentIdentColl: *MList*; { quantified variables, keeps spellings of vars }
  nSegmentPos: *Position*;
  nFirstSententialOperand: *FormulaPtr*;
  nRestriction: *FormulaPtr*;
  nAttrCollection: *MList*;
  nNoneOcc: *boolean*;
  nNonPos: *Position*;
  nPostNegated: *boolean*;
  nArgListNbr: *integer*; { position in a term (§1519) }
  nArgList: **array of** *arg_type*;
  nFunc: **array of** *func_type*;
  constructor *Init*;
  ⟨ Methods implemented by subclasses of *SubexpObj* 716 ⟩
end ;

This code is used in section 1199.

1389. The *TermNbr* is used to treat a list of terms as a stack data structure. Specifically, the *Term* array is treated as a stack, and the *TermNbr* is the index of the "top" of the stack.

⟨ Local variables for parser additions 1209 ⟩ +≡
TermNbr: *integer*;

1390. ⟨ Extended subexpression implementation 1390 ⟩ ≡
{ *Subexpressions handling* }
constructor *extSubexpObj.Init*;
const $MaxArgListNbr = 20$;
begin *inherited Init*; $nRestriction \leftarrow$ **nil**; $nTermBase \leftarrow TermNbr$; $nArgListNbr \leftarrow 0$;
$setlength(nArgList, MaxArgListNbr + 1)$; $setlength(nFunc, MaxArgListNbr + 1)$;
$nArgList[0].Start \leftarrow TermNbr + 1$;
end;

See also sections 1391, 1392, 1393, 1395, 1396, 1397, 1398, 1399, 1400, 1401, 1402, 1403, 1404, 1405, 1406, 1407, 1408, 1409, 1410, 1411, 1413, 1417, 1429, 1430, 1431, 1432, 1433, 1434, 1435, 1436, 1437, 1438, 1439, 1440, 1441, 1442, 1443, 1444, 1445, 1446, 1447, 1448, 1449, 1450, 1451, 1452, 1453, 1454, 1455, 1456, 1457, 1458, 1459, 1460, 1461, 1462, 1463, 1464, 1465, 1466, 1467, 1468, 1469, 1470, 1471, 1472, 1473, 1474, 1475, 1476, 1477, 1478, 1479, 1480, 1481, 1482, 1483, 1484, 1485, 1486, 1487, and 1488.

This code is used in section 1200.

1391. When the Parser is about to parse a stack of attributes, either in a registration or on a type, we need to initialize the appropriate state variables. We also need the caller's *nAttrCollection* to be initialized with an empty list.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.StartAttributes*;
begin $nAttrCollection.Init(0)$; $gLastType \leftarrow$ **nil**;
end;

1392. When the Parser expects an adjective, and the caller is used to store the adjective or attribute, we need to check if it is negated. This handles it.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.ProcessNon*;
begin $nNoneOcc \leftarrow CurWord.Kind = sy\_Non$; $nNonPos \leftarrow CurPos$;
end;

1393. Pop arguments from term stack. This will take some parameter *aBase* and copy pointers to each element of $Term[aBase \mathrel{..} TermNbr]$ 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 $aBase \le 0$, this will set *TermNbr* to a negative number.

⟨ Extended subexpression implementation 1390 ⟩ +≡
function $CreateArgs(aBase : integer)$: *PList*;
var k: *integer*; *lList*: *PList*;
begin $lList \leftarrow new(PList, Init(TermNbr - aBase))$;
for $k \leftarrow aBase$ **to** *TermNbr* **do** $lList.Insert(Term[k])$;
$TermNbr \leftarrow aBase - 1$; $CreateArgs \leftarrow lList$;
end;

1394. The "process (singular) attribute" method is invoked in the "process (plural) attributes" procedure (§1524), and in the *ATTSubexpression* procedure (§1641). 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.

⟨ Global variables introduced in `parseraddition.pas` 1202 ⟩ +≡
gLastAdjective: *AdjectiveExpressionPtr*;

1395. ⟨ Extended subexpression implementation 1390 ⟩ +≡

```
procedure extSubexpObj.ProcessAttribute;
  var lFormatNr: integer;
  begin if CurWord.Kind = AttributeSymbol then
    begin
        lFormatNr ← gFormatsColl.LookUp_PrefixFormat('V', CurWord.Nr, TermNbr − nTermBase + 1);
    if lFormatNr = 0 then   { format not found! }
      begin gLastAdjective ← new(AdjectivePtr, Init(CurPos, 0, CreateArgs(nTermBase + 1)));
      Error(CurPos, 175)
      end
    else begin
          gLastAdjective ← new(AdjectivePtr, Init(CurPos, CurWord.Nr, CreateArgs(nTermBase + 1)));
      if nNoneOcc then gLastAdjective ← new(NegatedAdjectivePtr, Init(nNonPos, gLastAdjective));
      end;
    end
  else   { needed for ATTSubexpression adjective cluster handling }
  begin gLastAdjective ← new(AdjectivePtr, Init(CurPos, 0, CreateArgs(nTermBase + 1)));
  end;
  nAttrCollection.Insert(gLastAdjective);
  end;
```

1396. These next next method is invoked before the Parser parses arguments for an attribute.

⟨ Extended subexpression implementation 1390 ⟩ +≡

```
procedure extSubexpObj.StartAttributeArguments;
  begin nTermBase ← TermNbr;
  end;
```

1397. The next two methods are invoked after the Parser has finished parsing the arguments for an attribute.

I am confused why there is duplicate code here, and the naming conventions suggest the *FinishAttributeArguments* method should be preferred.

⟨ Extended subexpression implementation 1390 ⟩ +≡

```
procedure extSubexpObj.CompleteAttributeArguments;
  begin nSubexpPos ← CurPos; nRightArgBase ← TermNbr;
  end;
procedure extSubexpObj.FinishAttributeArguments;
  begin nSubexpPos ← CurPos; nRightArgBase ← TermNbr;
  end;
```

1398. This allocates a new list of pointers, moves the caller's *nAttrCollection* into the list, and updates the *gAttrColl* state variable to point at them.

Again, this should be named *FinishedAdjectiveCluster* to be consistent with the naming conventions seemingly adopted.

⟨ Extended subexpression implementation 1390 ⟩ +≡

```
procedure extSubexpObj.CompleteAdjectiveCluster;
  begin gAttrColl ← new(PList, MoveList(nAttrCollection));
  end;
```

1399. When the Parser works its way through a registration block, check that the *TermNbr* points to not farther ahead than one more token ahead from the caller's *nTermBase* field. Raise an error if that happens.

This method is only invoked in the Parser module's the *RegisterCluster* (§1643) procedure.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.CompleteClusterTerm*;
  begin if $TermNbr - nTermBase > 1$ **then**
    begin *ErrImm*(379); *gLastTerm* ← *new*(*IncorrectTermPtr*, *Init*(*CurPos*));
    end;
  end;

1400. A "simple term" appears to be a variable. This is used when the Parser parses an identifier as a closed term (§1509). The state variable *gLastTerm* is updated to point to a newly allocated *SimpleTerm* object.

This method should probably be moved closer to the other methods used when parsing terms.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.ProcessSimpleTerm*;
  begin *gLastTerm* ← *new*(*SimpleTermPtr*, *Init*(*CurPos*, *GetIdentifier*));
  end;

1401. Qualified terms. The Parser invokes *ProcessQua* when it is looking directly at a "`qua`" token, specifically in the *AppendQua* (§1504) procedure. The *ProcessQua* method is used nowhere else. It is solely responsible for "marking the current position" of the Parser, and storing that in the caller's *nQuaPos* field.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.ProcessQua*;
  begin *nQuaPos* ← *CurPos*
  end;

1402. The Parser invokes the *FinishedQualifiedTerm* method after encountering a "`qua`" and after parsing the type. This method constructs a new *QualifiedTerm* object reflecting the top of the *Term* stack is taken "`qua`" the *gLastType*, and the mutates the top of the *Term* stack to be this newly allocated *QualifiedTerm* object.

This method does not push anything new to the term stack, but it does mutate the *Term* stack.

This method is used nowhere else other than the Parser's *AppendQua* (§1504) procedure.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishQualifiedTerm*;
  begin *Term*[*TermNbr*] ← *new*(*QualifiedTermPtr*, *Init*(*nQuaPos*, *Term*[*TermNbr*], *gLastType*));
  end;

1403. Although the "`exactly`" reserved keyword is not used for anything, the method for *ProcessExactly* marks the current position and stores it in the caller's *nQuaPos*, then *updates* (**not** pushes) to the top of the term stack by turning the top of the stack into an *ExactlyTerm* object.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.ProcessExactly*;
  begin *nQuaPos* ← *CurPos*; *Term*[*TermNbr*] ← *new*(*ExactlyTermPtr*, *Init*(*nQuaPos*, *Term*[*TermNbr*]));
  end;

1404. Arguments to a term. The *CheckTermLimit* procedure is a "private helper function" for the *FinishArgument* method.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *CheckTermLimit*;
  var *l*: *integer*;
  begin if $TermNbr \geq length(Term)$ **then**
    begin $l \leftarrow 2 * length(Term)$; *setlength*(*Term*, *l*);
    end;
  end;

1405. Pushing the Term stack. This method pushes the *gLastTerm* state variable's contents to the *Term* stack, mutating the *TermNbr* and *Term* module-local variables.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishArgument*;
  begin *CheckTermLimit*; *inc*(*TermNbr*); $Term[TermNbr] \leftarrow gLastTerm$;
  end;

1406. Pop the Term stack. The evil twin to "pushing" an element onto a stack, "popping" a stack removes the top element. We pop the *Term* stack whenever we finish the term.

This is only used in *AppendFunc* (§1519).

This should probably check that the *Term* stack is not empty before being invoked.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishTerm*;
  begin $gLastTerm \leftarrow Term[TermNbr]$; *dec*(*TermNbr*);
  end;

Subsection 23.3.1. Parsing Types

1407. When we start parsing a new type, we make sure the *gLastType* state variable is not caching an old type. We assign it to be the **nil** pointer.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.StartType*;
  begin $gLastType \leftarrow$ **nil**;
  end;

1408. This is invoked only by the Parser's *RadixTypeSubexpression* (§1525) procedure. The Parser delegates the work of storing the mode information to this method. In turn, the caller's *nModeKind* field stores the current word's token *Kind*, and the caller's *nModeNr* field stores the current word's number. The Parser's current position is marked and stored in the caller's *nSubexpPos* field.

But no state variables are mutated by this method.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.ProcessModeSymbol*;
    begin $nModeKind \leftarrow CurWord.Kind$; $nModeNr \leftarrow CurWord.Nr$;
    if $(CurWord.Kind = sy\_Set)$ $\{?\wedge(AheadWord.Kind \neq sy\_Of)?\}$
    then $nModeKind \leftarrow ModeSymbol$; $nSubexpPos \leftarrow CurPos$;
    end ;

1409. The Parser has just finished parsing a type and its arguments — "⟨*Mode*⟩ `of` ⟨*Term-list*⟩" or "⟨*Structure*⟩ `over` ⟨*Term-list*⟩". 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 *gFormatsColl*, a 185 error will be raised.

This is invoked only by the Parser's *RadixTypeSubexpression* (§1525) procedure.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishType*;
  var *lFormatNr*: *integer*;
  begin case *nModeKind* **of**
  ModeSymbol: **begin**
      lFormatNr ← *gFormatsColl.LookUp_PrefixFormat*(`´M´`, *nModeNr*, *TermNbr* − *nTermBase*);
    if *lFormatNr* = 0 **then** *Error*(*nSubexpPos*, 151); { format missing }
    gLastType ← *new*(*StandardTypePtr*, *Init*(*nSubexpPos*, *nModeNr*, *CreateArgs*(*nTermBase* + 1)));
    end;
  StructureSymbol: **begin**
      lFormatNr ← *gFormatsColl.LookUp_PrefixFormat*(`´L´`, *nModeNr*, *TermNbr* − *nTermBase*);
    if *lFormatNr* = 0 **then** *SemErr*(185); { format missing }
    gLastType ← *new*(*StructTypePtr*, *Init*(*nSubexpPos*, *nModeNr*, *CreateArgs*(*nTermBase* + 1)));
    end;
  othercases begin *gLastType* ← *new*(*IncorrectTypePtr*, *Init*(*CurPos*)); **end**;
  endcases;
  end;

1410. If the Parser has the misfortune of trying to make sense of a malformed type expression, then with a heavy heart it invokes this method to update the *gLastType* state variable to be an incorrect type expression at the current position.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.InsertIncorrType*;
  begin *gLastType* ← *new*(*IncorrectTypePtr*, *Init*(*CurPos*));
  end;

1411. When the parser encounters a qualifying formula ("⟨*Term*⟩ `is` ⟨*Type*⟩") or is parsing a type for a cluster (the "`cluster` ...`for` ⟨*Type*⟩"), 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.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.CompleteType*;
  var *j*: *integer*;
  begin *mizassert*(5433, *gLastType* ≠ **nil**);
  if *nAttrCollection.Count* > 0 **then**
    begin *gLastType* ← *new*(*ClusteredTypePtr*, *Init*(*gLastType*↑.*nTypePos*, *new*(*PList*,
        Init(*nAttrCollection.Count*)), *gLastType*));
    for *j* ← 0 **to** *nAttrCollection.Count* − 1 **do**
      ClusteredTypePtr(*gLastType*)↑.*nAdjectiveCluster*↑.*Insert*(*PObject*(*nAttrCollection.Items*↑[*j*]));
    nAttrCollection.DeleteAll;
    end;
  end;

Subsection 23.3.2. Parsing operator precedence

1412. Mario Carneiro's "Mizar in Rust" (§6.2) 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* (§648) global variable, comes into play.

1413. Starting a "long term".
We can observe that *nTermBase* is initialized upon construction to *TermNbr*; in *ProcessAtomicFormula* and *StartPrivateFormula* it is assigned to *TermNbr*.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.StartLongTerm*;
  begin *nArgListNbr* ← 0; *nArgList*[0].*Length* ← *TermNbr* − *nTermBase*;
  end;

1414. Malformed term errors.
We should remind the reader, errors 165–175 are "unknown functor format", errors 176 is "unknown attribute format", and error 177 is "unknown structure format". Only when such an error occurs, the flow experiences a **goto** *AfterBalance*.

For an example of a 168, 169 error:

```
for x being Nat
holds (id + x +) = x;
```

For an example of a 170, 171 error (the first `0` will be flagged 170, the second `0` will be flagged as 171):

```
for x being Nat
holds 0 0 + x = x;
```

For an example of a 172, 173 error:

```
for x being Nat
holds x + / = x;
```

For an example of a 174, 175 error:

```
for x being Nat
holds x + (1,2) + x = x;
```

1415. We can recall that a "generic" term looks like an infixed operator of the form

$$(t_1^{(\ell)},\dots,t_m^{(\ell)})\ t\ (t_1^{(r)},\dots,t_n^{(r)})$$

The parentheses are optional. Constants will have $m = n = 0$ and look like () t (). Function-like terms will have $m = 0$ and look like () t $(t_1^{(r)},\dots,t_n^{(r)})$. The problem statement could be re-phrased as: given several infixed terms without parentheses inserted anywhere, determine how to cluster terms together.

1416. The problem statement for constructing the syntax tree for a term is something like the following: we have an expression of the form

$$x_1^{(0)},\dots,x_{k_0}^{(0)}\ F_1\ x_1^{(1)},\dots,x_{k_1}^{(1)}\ F_2\ \cdots\ F_n\ x_1^{(n)},\dots,x_{k_n}^{(n)}$$

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 $1\leq a\leq i\leq b\leq n$ with node F_i being the root of the subtree. There are $O(n^3)$ such subproblems, and they can be calculated from smaller subproblems in $O(n)$. This might seem alarmingly large, but usually the terms in Mizar are sufficiently small.

It is interesting to see how other languages tackle this problem, so I am going to give a haphazard literature review:

(1) Nils Anders Danielsson and Ulf Norell's "Parsing Mixfix Operators" (in SB. Scholz and O. Chitil (eds.), *Symposium on Implementation and Application of Functional Languages*, Springer 2008, pp. 80–99; doi:10.1007/978-3-642-24452-0_5) discuss how Agda approaches parsing mixfix operators with different precedence.

(2) The Isabelle proof assistant uses a modified version of Earley parsing of terms, supporting precedence between 0 to 1000.

1417. The only two place where *FinishLongTerm* is invoked is in the *AppendFunc* procedure (§1519) in `parser.pas`.

This relies on *MFormatsList.LookUpFuncFormat* (§667), which attempts to look up an *MInfixFormatObj* (§655) with a given id number as well as number of left and right arguments.

We will need to populate *ArgsLength* and *To_Right* to determine the syntax tree for the term (which is our real goal here). The *ArgsLength* encodes the number of terms are to the left and right of each "internal node". The *To_Right* controls associativity (which is how Mizar handles operator precedence): if node F_{k+1} is higher precedence than node F_k, then *To_Right*(k) is true.

The *Exchange*(i) procedure will make node i a child of $i-1$ (when node i is a child of $i-1$), and vice-versa. Visually, this means we transform the tree as:

$$\bigl(\cdots F_{i-1}\ x_1,\ldots,x_\ell\bigr), x_{\ell+1},\ldots,x_n\ F_i\cdots \longleftrightarrow \cdots F_{i-1}\ x_1,\ldots,x_{\ell-1}, \bigl(x_\ell, x_{\ell+1},\ldots,x_n\ F_i\cdots\bigr)$$

Observe that "*Exchange*(i); *Exchange*(i)" is equivalent to doing nothing.

We should recall (§1387) that *nArgList* is an array of "**record** *Instance*, *SymPri*: *integer*; *FuncPos*: *Position*; **end**".

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishLongTerm*;
  var *ArgsLength*: **array of record** *l*, *r*: *integer*;
      end;
    To_Right: **array of** *boolean*;
    procedure *Exchange*(*i* : *integer*);
      var *l*: *integer*;
      begin $l \leftarrow$ *ArgsLength*[*i*].*l*; *ArgsLength*[*i*].*l* $\leftarrow$ *ArgsLength*[$i-1$].*r*; *ArgsLength*[$i-1$].*r* $\leftarrow l$;
      To_Right[$i-1$] $\leftarrow \neg$ *To_Right*[$i-1$];
      end;
  var *Bl*, *new_Bl*: *integer*; { indexes *nFunc*, *ArgsLength* }
    i, *j*, *k*: *integer*; { various indices }
    ⟨ Variables for finishing a long term in a subexpression 1426 ⟩
  label *Corrected*, *AfterBalance*;
  begin ⟨ Rebalance the long term tree 1418 ⟩
AfterBalance: ⟨ Construct the term's syntax tree after balancing arguments among subterms 1427 ⟩
  end;

1418. Rebalancing the term tree.

Note that *nArgListNbr* is mutated only in *extSubexpObj.ProcessFunctorSymbol* (§1429), and in *ProcessAtomicFormula* (§1459) it is reset to zero.

define *missing_functor_format* $\equiv$ *gFormatsColl.LookUp_FuncFormat*(*Instance*, l, r) $= 0$

⟨ Rebalance the long term tree 1418 ⟩ $\equiv$
  ⟨ Initialize *To_Right* and *ArgsLength* arrays 1421 ⟩
  ⟨ Initialize *Bl*, **goto** *AfterBalance* if term has at most one argument 1423 ⟩
    $\{ Bl = 1 \lor Bl = 2 \}$
  for $k \leftarrow 2$ **to** $nArgListNbr - 1$ **do**
    with *nFunc*[k], *ArgsLength*[k] **do**
      begin if *missing_functor_format* **then** ⟨ Guess the k^{th} functor format 1424 ⟩
    Corrected: **end**;
  for $j \leftarrow nArgListNbr$ **downto** $Bl + 1$ **do**
    with *nFunc*[j], *ArgsLength*[j] **do**
      begin if $\neg$*missing_functor_format* **then goto** *AfterBalance*;
      Exchange(j); ⟨ Check for 172/173 error, **goto** *AfterBalance* if erred 1419 ⟩
      end;
  ⟨ Check for 174/175 error, **goto** *AfterBalance* if erred 1420 ⟩

This code is used in section 1417.

1419. ⟨ Check for 172/173 error, **goto** *AfterBalance* if erred 1419 ⟩ $\equiv$
  if *missing_functor_format* **then**
    begin *Error*(*FuncPos*, 172); *Error*(*nFunc*[*nArgListNbr*].*FuncPos*, 173); **goto** *AfterBalance*; **end**;

This code is used in section 1418.

1420. ⟨ Check for 174/175 error, **goto** *AfterBalance* if erred 1420 ⟩ $\equiv$
  with *nFunc*[*Bl*], *ArgsLength*[*Bl*] **do**
    if *missing_functor_format* **then**
      begin *Error*(*FuncPos*, 174); *Error*(*nFunc*[*nArgListNbr*].*FuncPos*, 175); **goto** *AfterBalance*; **end**;

This code is used in section 1418.

1421. We first allocate the arrays, the we initialize the values.

⟨ Initialize *To_Right* and *ArgsLength* arrays 1421 ⟩ $\equiv$
  setlength(*ArgsLength*, $nArgListNbr + 1$); *setlength*(*To_Right*, $nArgListNbr + 1$);
  setlength(*Depo*, $nArgListNbr + 1$);

See also section 1422.

This code is used in section 1418.

1422. The initial guess depends on whether F_k has precedence over F_{k+1} or not.

If F_{k+1} has higher precedence than F_k, then the initial guess groups terms as:

$$\cdots F_k \ \bigl((x_1^{(k)}, \ldots, x_{m_k}^{(k)}) F_{k+1}(\cdots)\bigr) \cdots, \quad \text{and} \quad \mathit{To\_Right}[k] = \mathit{true}.$$

On the other hand, if F_{k+1} *does not* have higher precedence than F_k, then we guess the terms are grouped as

$$\cdots \bigl(\cdots F_k(x_1^{(k)}, \ldots, x_{m_k}^{(k)})\bigr) \ F_{k+1} \cdots, \quad \text{and} \quad \mathit{To\_Right}[k] = \mathit{false}.$$

This is a first stab, but sometimes we get lucky and it's correct.

define *next_term_has_higher_precedence*(#) ≡
  gPriority.*Value*(*ord*(´O´), *nFunc*[#].*Instance*) < *gPriority*.*Value*(*ord*(´O´), *nFunc*[# + 1].*Instance*)

⟨ Initialize *To_Right* and *ArgsLength* arrays 1421 ⟩ +≡
  ArgsLength[1].*l* ← *nArgList*[0].*Length*; *To_Right*[0] ← *true*;
  for *k* ← 1 **to** *nArgListNbr* − 1 **do**
    with *ArgsLength*[*k*] **do**
      if *next_term_has_higher_precedence*(*k*) **then**
        begin *r* ← 1; *ArgsLength*[*k* + 1].*l* ← *nArgList*[*k*].*Length*; *To_Right*[*k*] ← *true* **end**
      else begin *r* ← *nArgList*[*k*].*Length*; *ArgsLength*[*k* + 1].*l* ← 1; *To_Right*[*k*] ← *false* **end**;
  ArgsLength[*nArgListNbr*].*r* ← *nArgList*[*nArgListNbr*].*Length*; *To_Right*[*nArgListNbr*] ← *false*;

1423. The first situation we encounter is if the user tries to tell Mizar to evaluate something like:

```
for x being Nat
holds x + (1,2) = x;
```

Mizar will not understand "`x + (1,2)`" because it is an invalid functor format — the format would look something like ⟨"+", left : 1, right : 1⟩ but the format of the expression is ⟨left : 1, right : 2⟩. The mismatch on the "right" values in the formats will raise a 165 error.

For a 166 error example,

```
for x being Nat
holds + / = x;
```

Mizar will not like the leading "`+ /`" expression, and flag this with the 166 error.

Mizar will flag "`+ 0`" as a 165 error.

⟨ Initialize *Bl*, **goto** *AfterBalance* if term has at most one argument 1423 ⟩ ≡
  with *nFunc*[1], *ArgsLength*[1] **do**
    begin if *nArgListNbr* = 1 **then**
      begin if *missing_functor_format* **then**
        begin *Error*(*FuncPos*, 165); **goto** *AfterBalance* **end**;
      goto *AfterBalance*;
      end;
    Bl ← 1;
    if *missing_functor_format* **then**
      begin *Exchange*(2); *Bl* ← 2;
      if *missing_functor_format* **then**
        begin *Error*(*FuncPos*, 166); **goto** *AfterBalance* **end**;
      end;
    end;

This code is used in section 1418.

1424. ⟨ Guess the k^{th} functor format 1424 ⟩ ≡

```
begin Exchange(k + 1); new_Bl ← 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 ← k;
  if missing_functor_format then
    begin Exchange(k + 1); new_Bl ← k + 1;
    if missing_functor_format then
      begin Error(FuncPos, 167); goto AfterBalance end;
    end;
  for j ← 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;
      end;
  ⟨ Check term Bl has valid functor format, goto AfterBalance if not 1425 ⟩
  end;
Bl ← new_Bl;
end;
```

This code is used in section 1418.

1425. ⟨ Check term *Bl* has valid functor format, **goto** *AfterBalance* if not 1425 ⟩ ≡

```
with nFunc[Bl], ArgsLength[Bl] do
  if missing_functor_format then
    begin Error(FuncPos, 170); Error(nFunc[k].FuncPos, 171); goto AfterBalance; end;
```

This code is used in section 1424.

1426. Constructing the syntax tree. The second half of finishing a long term constructs the syntax tree for the term.

⟨ Variables for finishing a long term in a subexpression 1426 ⟩ ≡

```
ak, pl, ll, kn: integer;
lTrm: TermPtr;
lLeftArgs, lRightArgs: PList;
DepoNbr: integer;
Depo: array of record FuncInstNr: integer;
    dArgList: PList;
    end;
```

This code is used in section 1417.

1427. ⟨ Construct the term's syntax tree after balancing arguments among subterms 1427 ⟩ ≡
⟨ Initialize symbol priorities, determine last *ll*, *pl* values 1428 ⟩

```
DepoNbr ← 0;
for kn ← nArgListNbr 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 ← CreateArgs(nArgList[kn].Start);   { (§1393) }
      lLeftArgs ← CreateArgs(nArgList[kn − 1].Start);
      lTrm ← new(InfixTermPtr, Init(FuncPos, Instance, lLeftArgs, lRightArgs));
      end;
    for j ← DepoNbr downto 1 do
      with Depo[j], nFunc[FuncInstNr] do
        begin if symPri ≤ nFunc[kn − 1].SymPri then break;
        dec(DepoNbr); lLeftArgs ← new(PList, Init(1)); lLeftArgs↑.Insert(lTrm);
        lTrm ← new(InfixTermPtr, Init(FuncPos, Instance, lLeftArgs, dArgList));
        end;
    gLastTerm ← lTrm;
    gSubexpPtr↑.FinishArgument;
    end
  else begin inc(DepoNbr);
    with Depo[DepoNbr] do
      begin FuncInstNr ← kn; dArgList ← CreateArgs(nArgList[kn].Start); end;
    end;
with nFunc[1] do
  begin lRightArgs ← CreateArgs(nArgList[1].Start); lLeftArgs ← CreateArgs(nArgList[0].Start);
  lTrm ← new(InfixTermPtr, Init(FuncPos, Instance, lLeftArgs, lRightArgs));
  end;
for j ← DepoNbr downto 1 do
  with Depo[j], nFunc[FuncInstNr] do
    begin lLeftArgs ← new(PList, Init(1)); lLeftArgs↑.Insert(lTrm);
    lTrm ← new(InfixTermPtr, Init(FuncPos, Instance, lLeftArgs, dArgList));
    end;
gLastTerm ← lTrm;
```

This code is used in section 1417.

1428. ⟨ Initialize symbol priorities, determine last *ll*, *pl* values 1428 ⟩ ≡

```
for ak ← 1 to nArgListNbr do
  begin ll ← 1; pl ← 1;
  if To_Right[ak − 1] then ll ← nArgList[ak − 1].Length;
  if ¬To_Right[ak] then pl ← nArgList[ak].Length;
  with nFunc[ak] do
    begin symPri ← gPriority.Value(ord('O'), Instance); end;
  end;
```

This code is used in section 1427.

Subsection 23.3.3. Processing subexpressions

1429. Note that *ProcessFunctorSymbol* is the only place where *nArgListNbr* is incremented. Processing functor symbols occurs in the parser's *AppendFunc* (§1519) in a loop.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.ProcessFunctorSymbol*;
  var *l*: *integer*;
  begin *inc*(*nArgListNbr*);
  if $nArgListNbr \geq length(nFunc)$ **then**
    begin $l \leftarrow 2 * length(nFunc) + 1$; *setlength*(*nArgList*, *l*); *setlength*(*nFunc*, *l*);
    end;
  $nArgList[nArgListNbr].Start \leftarrow TermNbr + 1$; $nFunc[nArgListNbr].FuncPos \leftarrow CurPos$;
  $nFunc[nArgListNbr].Instance \leftarrow CurWord.Nr$;
  end;

1430. The Parser is in the middle of *AppendFunc* and has just finished parsing a term t or a tuple of terms `(` t_1, ..., 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).

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishArgList*;
  begin $nArgList[nArgListNbr].Length \leftarrow TermNbr - nArgList[nArgListNbr].Start + 1$;
  end;

1431. The Parser is looking at "`where`" or (when the variables are all reserved) a colon "`:`", the Parser invokes the *StartFraenkelTerm* which will store the previous term in the *nSample* field — so schematically, the Fraenkel term could look like

$$\{\langle nSample\rangle \texttt{ where } \langle Postqualification\rangle : \langle Formula\rangle\}$$

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.StartFraenkelTerm*;
  begin $nSample \leftarrow gLastTerm$;
  end;

1432. This is only invoked in the Parser's *ProcessPostqualification* (§1506) procedure, which is only invoked after the Parser calls the *extSubexp* object's *StartFraenkelTerm* method.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.StartPostqualification*;
  begin *nPostQualList.Init*(0);
  end;

1433. The parser is looking at the post-qualified segment of a Fraenkel operator. This will be a list of variables "`being`" a type, we allocate an array for the variables. This is handled by the *StartPostQualifyingSegment* method.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.StartPostQualifyingSegment*;
  begin *nSegmentIdentColl.Init*(2);
  end;

1434. While looping over the comma-separated list of variables in a post-qualified segment (in a Fraenkel term), the Parser invokes the *ProcessPostqualifiedVariable* on each iteration until it has parsed all the variables. This allocates a new *Variable* object, and pushes it onto the *nSegmentIdentColl* "stack".

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.ProcessPostqualifiedVariable*;
  begin *nSegmentIdentColl.Insert*(*new*(*VariablePtr*, *Init*(*CurPos*, *GetIdentifier*)));
  end;

1435. The parser is looking at "`is`" or "`are`" in a Fraenkel term's post-qualification segment, but has not yet parsed the type. This method will assign the *nSegmentPos* field to be the current position, and assign the *gLastType* state variable to be the **nil** pointer.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.StartPostqualificationSpecyfication*;
  begin *nSegmentPos* ← *CurPos*; *gLastType* ← **nil**;
  end;

1436. The Parser has just parsed either (1) a comma-separated list of variables, the copula "`is`" or "`are`", and the type; or (2) a comma-separated list of reserved variables (but no copula and no type). We just need to construct an appropriate node for the abstract syntax tree. This method will append a new Segment to the *nPostQualList*.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishPostQualifyingSegment*;
  var *k*: *integer*; *lSegment*: *ExplicitlyQualifiedSegmentPtr*;
  begin if *gLastType* ≠ **nil then**
    begin *lSegment* ← *new*(*ExplicitlyQualifiedSegmentPtr*, *Init*(*nSegmentPos*, *new*(*PList*, *Init*(0)),
        gLastType)); *nPostQualList.Insert*(*lSegment*);
    for *k* ← 0 **to** *nSegmentIdentColl.Count* − 1 **do**
      begin *ExplicitlyQualifiedSegmentPtr*(*lSegment*)↑.*nIdentifiers.Insert*(*nSegmentIdentColl.Items*↑[*k*]);
      end;
    end
  else begin for *k* ← 0 **to** *nSegmentIdentColl.Count* − 1 **do**
      begin *nPostQualList.Insert*(*new*(*ImplicitlyQualifiedSegmentPtr*,
          Init(*VariablePtr*(*nSegmentIdentColl.Items*↑[*k*])↑.*nVarPos*, *nSegmentIdentColl.Items*↑[*k*])));
      end;
    end;
  nSegmentIdentColl.DeleteAll; *nSegmentIdentColl.Done*;
  end;

1437. The Parser has just finished the formula in a Fraenkel term, and it is staring at the closet "}" bracket. The Parser invokes this method to construct a new *FraenkelTerm* AST node, and updates the *gLastTerm* to point at it.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishFraenkelTerm*;
  begin *gLastTerm* ← *new*(*FraenkelTermPtr*, *Init*(*CurPos*, *new*(*PList*, *MoveList*(*nPostQualList*)),
      nSample, *gLastFormula*));
  end;

1438. The Parser has already encountered "`the set`" and the next token is "`of`", which means the Parser has encountered a "simple" Fraenkel term of the form "`the set of all` ⟨*Term*⟩...". This method will be invoked once the Parser has stumbled across the "`all`". The caller updates its *nAllPos* to the Parser's current position.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.StartSimpleFraenkelTerm*;
  begin *nAllPos* ← *CurPos*;
  end;

1439. The Parser has just finished parsing the post-qualification to the simple Fraenkel term, which means it has finished parsing the simple Fraenkel term. This method allocates a new *SimpleFraenkelTerm* AST node with the accumulated AST nodes, then updates the *gLastTerm* to point to the allocated *SimpleFraenkelTerm* node.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishSimpleFraenkelTerm*;
  begin *gLastTerm* ← *new*(*SimpleFraenkelTermPtr*, *Init*(*nAllPos*, *new*(*PList*, *MoveList*(*nPostQualList*)),
      nSample));
  end;

1440. The Parser is looking at a closed term of the form "⟨*Identifier*⟩ `(`...", and so it looks like a private functor. This method updates the caller's *nSubexpPos* to the Parser's current position, and the *nSpelling* is assigned to the identifier's number (for the private functor).

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.StartPrivateTerm*;
  begin *nSubexpPos* ← *CurPos*; *nSpelling* ← *CurWord.Nr*;
  end;

1441. The Parser just finished parsing all the arguments to the private functor, and is looking at the closing parentheses for the private functor. This method allocates a new *PrivateFunctorTerm* object, using the arguments just parsed, and updates the *gLastTerm* state variable to point to it.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishPrivateTerm*;
  begin *gLastTerm* ← *new*(*PrivateFunctorTermPtr*, *Init*(*nSubexpPos*, *nSpelling*,
      CreateArgs(*nTermBase* + 1)));
  end;

1442. The Parser has just encountered either a left bracket term or the opening left bracket for a set "{". The Parser calls this method, which just updates the caller's *nSymbolNr* to be whatever the current token's numeric ID value is.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.StartBracketedTerm*;
  begin *nSymbolNr* ← *CurWord.Nr*;
  end;

1443. If the Parser is in panic mode, this method does nothing.

Either the Parser has finished parsing an enumerated set $\{x_1, \ldots, x_n\}$ or a bracketed term. We need to double check the format for the bracket matches what is stored in the *gFormatsColl*, and raise a 152 error if there's a mismatch. Otherwise, allocate a new AST node for the bracketed term, and use *CreateArgs* on the terms contained within the brackets.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishBracketedTerm*;
  var *lFormatNr*: *integer*;
  begin if *StillCorrect* **then**
    begin *nRSymbolNr* ← *CurWord.Nr*; *lFormatNr* ← *gFormatsColl.LookUp_BracketFormat*(*nSymbolNr*,
        nRSymbolNr, *TermNbr* − *nTermBase*, 0, 0);
    if *lFormatNr* = 0 **then** *SemErr*(152);
    gLastTerm ← *new*(*CircumfixTermPtr*, *Init*(*CurPos*, *nSymbolNr*, *nRSymbolNr*,
        CreateArgs(*nTermBase* + 1)));
    end;
  end;

1444. Remember that Mizar calls "an instance of structure" an **"Aggregate"**. When the Parser is parsing for a closed subterm and has stumbled across a structure constructor (§1510), it first invokes this method. This stores the ID number for the structure in the caller's *nSymbolNr*.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.StartAggregateTerm*;
  begin *nSymbolNr* ← *CurWord.Nr*;
  end;

1445. The Parser has just parsed the arguments for the structure constructor, and the Parser is now looking at the "`#)`" token. This method is invoked.

We should check the format for the structure constructor is stored in the *gFormatsColl*. If not, raise a 176 error. Otherwise, we allocate a new *AggregateTerm* with the parsed arguments, and then update the *gLastTerm* pointer to point at it.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishAggregateTerm*;
  var *lFormatNr*: *integer*;
  begin *lFormatNr* ← *gFormatsColl.LookUp_PrefixFormat*(´G´, *nSymbolNr*, *TermNbr* − *nTermBase*);
  if *lFormatNr* = 0 **then** *Error*(*CurPos*, 176); { missing format error }
  gLastTerm ← *new*(*AggregateTermPtr*, *Init*(*CurPos*, *nSymbolNr*, *CreateArgs*(*nTermBase* + 1)));
  end;

1446. The Parser is parsing for a closed subterm, and has stumbled across "`the`" and is looking at a selector token (§1515). 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.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.StartSelectorTerm*;
  begin *nSymbolNr* ← *CurWord.Nr*; *nSubexpPos* ← *CurPos*; *nNextWord* ← *AheadWord.Kind*;
  end;

1447. The Parser has just parsed "`the` ⟨*Selector*⟩ `of` ⟨*Term*⟩". 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.

⟨ Extended subexpression implementation 1390 ⟩ +≡

procedure *extSubexpObj.FinishSelectorTerm*;
  var *lFormatNr*: *integer*;
  begin *lFormatNr* ← *gFormatsColl.LookUp_PrefixFormat*(´U´, *nSymbolNr*, 1);
  if *lFormatNr* = 0 **then** *Error*(*nSubexpPos*, 182); { missing format error }
  if *nNextWord* = *sy_Of* **then**
    gLastTerm ← *new*(*SelectorTermPtr*, *Init*(*nSubexpPos*, *nSymbolNr*, *gLastTerm*))
  else if *in_AggrPattern* **then**
      gLastTerm ← *new*(*InternalSelectorTermPtr*, *Init*(*nSubexpPos*, *nSymbolNr*))
    else begin *gLastTerm* ← *new*(*IncorrectTermPtr*, *Init*(*nSubexpPos*)); *Error*(*nSubexpPos*, 329)
      end;
  end;

1448. The Parser is about to start parsing a forgetful functor (§1516) — 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`".

⟨ Extended subexpression implementation 1390 ⟩ +≡

procedure *extSubexpObj.StartForgetfulTerm*;
  begin *nSymbolNr* ← *CurWord.Nr*; *nSubexpPos* ← *CurPos*; *nNextWord* ← *AheadWord.Kind*;
  end;

1449. The Parser just finished parsing a forgetful functor. If the Parser is not panicking, check the format for the forgetful functor matches what is stored in the *gFormatsColl* state variable. If the format is invalid, raise a 184 error.

Whether the Parser is panicking or not, allocate a new *ForgetfulFunctor* term, and update the *gLastTerm* to point to it.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishForgetfulTerm*;
  var *lFormatNr*: *integer*;
  begin *lFormatNr* ← 0;
  if *StillCorrect* **then**
    begin *lFormatNr* ← *gFormatsColl.LookUp_PrefixFormat*(´J´, *nSymbolNr*, 1);
    if *lFormatNr* = 0 **then** *Error*(*nSubexpPos*, 184); { missing format }
    end;
  gLastTerm ← *new*(*ForgetfulFunctorTermPtr*, *Init*(*nSubexpPos*, *nSymbolNr*, *gLastTerm*));
  end;

1450. There are several situations where this is invoked:

(1) The Parser has just parsed "`the`" but is not looking at a selector symbol ("`the multF of`..."), nor is the Parser looking at a forgetful functor ("`the multMagma of`..."). Then this is interpreted as looking at a choice operator (§1515).
(2) The Parser has just parsed "`the`" but is not looking at a forgetful functor, so the Parser believes it must be looking at a choice operator (§1516).
(3) The Parser has just parsed "`the`" and is now looking at "`set`" — so this is invoking the axiom of choice to pick "`the set`" (§1517).

In these three situations, the Parser invokes this method. It just updates the caller's *nSubexpPos* field to point to the Parser's current position.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.StartChoiceTerm*;
  begin *nSubexpPos* ← *CurPos*;
  end;

1451. The Parser has just parsed a type, and now believes it has finished parsing a choice expression. Then it invokes this method to construct an appropriate AST node for the term, by specifically allocating a new *ChoiceTerm* for the *gLastType* type. We then update the *gLastTerm* state variable to point to this newly allocated term.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishChoiceTerm*;
  begin *gLastTerm* ← *new*(*ChoiceTermPtr*, *Init*(*nSubexpPos*, *gLastType*));
  end;

1452. When the Parser encounters a numeral while seeking a closed subterm (§1508), it invokes this method to allocate a new *NumeralTerm*. The *gLastTerm* state variable is updated to point to this newly allocated numeral object.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.ProcessNumeralTerm*;
  begin *gLastTerm* ← *new*(*NumeralTermPtr*, *Init*(*CurPos*, *CurWord.Nr*));
  end;

1453. The Parser tries to parse a closed subterm (§1508) 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.

⟨Extended subexpression implementation 1390⟩ +≡
procedure *extSubexpObj.ProcessItTerm*;
  begin if *it_Allowed* **then** *gLastTerm* ← *new*(*ItTermPtr*, *Init*(*CurPos*))
  else begin *gLastTerm* ← *new*(*IncorrectTermPtr*, *Init*(*CurPos*)); *ErrImm*(251)
    end;
  end;

1454. 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 1390⟩ +≡
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;

1455. 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.

⟨Extended subexpression implementation 1390⟩ +≡
procedure *extSubexpObj.InsertIncorrTerm*;
  begin *gLastTerm* ← *new*(*IncorrectTermPtr*, *Init*(*CurPos*));
  end;

Subsection 23.3.4. Parsing formulas

1456. The Parser is trying to parse an atomic formula (§1544), but something has gone awry. Allocate a new *IncorrectFormula* object located at the Parser's current position, update the *gLastFormula* state variable to point to it, and "reset" the *TermNbr* state variable to point to where the caller's *nTermBase* is located.

⟨Extended subexpression implementation 1390⟩ +≡
procedure *extSubexpObj.InsertIncorrBasic*;
  begin *gLastFormula* ← *new*(*IncorrectFormulaPtr*, *Init*(*CurPos*)); *TermNbr* ← *nTermBase*;
  end;

1457. While the Parser was trying to parse a formula, it found something which "doesn't quite fit". Allocate a new *IncorrectFormula* object, then update the *gLastFormula* state variable to point to it.

⟨Extended subexpression implementation 1390⟩ +≡
procedure *extSubexpObj.InsertIncorrFormula*;
  begin *gLastFormula* ← *new*(*IncorrectFormulaPtr*, *Init*(*CurPos*));
  end;

1458. If we are in a proof, allocate a new *ThesisFormula* object (recall the `WEB` macro for this §**????**). Otherwise, raise a 65 error.

⟨Extended subexpression implementation 1390⟩ +≡
procedure *extSubexpObj.ProcessThesis*;
  begin if $gProofCnt > 0$ **then** $gLastFormula \leftarrow thesis\_formula$
  else begin *ErrImm*(65); $gLastFormula \leftarrow new(IncorrectFormulaPtr, Init(CurPos))$;
    end;
  end;

1459. The Parser has encountered "⟨*Term*⟩ `is`", or some other generic atomic formula (§1544), this method is invoked.

If more than one term appears before the "`is`" token (i.e., if $TermNbr - nTermBase \neq 1$), then a 157 error is raised. There is a Polish comment here, "Trzeba chyba wstawic recovery dla $TermNbr = nTermBase$", which I translated to English.

This will initialize the fields for the caller in preparation for parsing some atomic formula. In particular, this is the only place where *TermNbr* is initialized to a nonzero value (and isn't in an incorrect formula).

⟨Extended subexpression implementation 1390⟩ +≡
procedure *extSubexpObj.ProcessAtomicFormula*;
  const $MaxArgListNbr = 20$;
  begin $nSubexpPos \leftarrow CurPos$; $nSymbolNr \leftarrow 0$;
  case *CurWord.Kind* **of**
  sy_Is: **if** $TermNbr - nTermBase \neq 1$ **then**
      begin *ErrImm*(157); $TermNbr \leftarrow nTermBase$; *InsertIncorrTerm*; *FinishArgument*;
        { I think you need to insert recovery for $TermNbr = nTermBase$ }
      end;
  endcases;
  $nRightArgBase \leftarrow TermNbr$; $nTermBase \leftarrow TermNbr$; $nPostNegated \leftarrow false$; $nArgListNbr \leftarrow 0$;
  $nArgList[0].Start \leftarrow TermNbr + 1$;
  end;

1460. The Parser is either finishing a "predicative formula" (§1543) or it's parsing a predicate pattern (§1612), it invokes this method to initialize the fields needed when forming an AST node. Specifically, the *nSubexpPos* is assigned to the Parser's current position, the *nSymbolNr* is updated either to the current token's ID number (if the current token is "`=`" or a predicate) or else assigned to be zero. Last, the *nRightArgBase* is assigned to equal the *TermNbr* state variable.

⟨Extended subexpression implementation 1390⟩ +≡
procedure *extSubexpObj.ProcessPredicateSymbol*;
  begin $nSubexpPos \leftarrow CurPos$;
  case *CurWord.Kind* **of**
  sy_Equal, *PredicateSymbol*: $nSymbolNr \leftarrow CurWord.Nr$;
  othercases $nSymbolNr \leftarrow 0$;
  endcases;
  $nRightArgBase \leftarrow TermNbr$;
  end;

1461. The Parser is parsing a "predicate formula" which has arguments on the righthand side of the predicate symbol (§1539).

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.ProcessRightSideOfPredicateSymbol*;
  begin *nRightSideOfPredPos* ← *CurPos*;
  case *CurWord.Kind* **of**
  sy_Equal, *PredicateSymbol*: *nSymbolNr* ← *CurWord.Nr*;
  othercases *nSymbolNr* ← 0;
  endcases;
  nRightArgBase ← *TermNbr*;
  end;

1462. The Parser has just finished a "predicate formula" (§1543), then this method is invoked to construct an AST for the formula. First we check if the format is valid. If the format for the formula is not found in the *gFormatsColl*, 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.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishPredicativeFormula*;
  var *lLeftArgs*, *lRightArgs*: *PList*; *lFormatNr*: *integer*;
  begin *lFormatNr* ← *gFormatsColl.LookUp_PredFormat*(*nSymbolNr*, *nRightArgBase* − *nTermBase*,
    TermNbr − *nRightArgBase*);
  if *lFormatNr* = 0 **then** *Error*(*nSubexpPos*, 153); { missing format }
  lRightArgs ← *CreateArgs*(*nRightArgBase* + 1); *lLeftArgs* ← *CreateArgs*(*nTermBase* + 1);
  gLastFormula ← *new*(*PredicativeFormulaPtr*, *Init*(*nSubexpPos*, *nSymbolNr*, *lLeftArgs*, *lRightArgs*));
  end;

1463. The Parser tries to construct an AST when finishing up the right-hand side of a predicative formula (§1539), it invokes this method after the *extSubexpObj.FinishPredicativeFormula* has been invoked.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishRightSideOfPredicativeFormula*;
  var *lRightArgs*: *PList*; *lLeftArgsNbr*, *lFormatNr*: *integer*; *lFrm*: *FormulaPtr*;
  begin *lFrm* ← *gLastFormula*;
  if *lFrm*↑.*nFormulaSort* = *wsNegatedFormula* **then** *lFrm* ← *NegativeFormulaPtr*(*lFrm*)↑.*nArg*;
  lLeftArgsNbr ← *RightSideOfPredicativeFormulaPtr*(*lFrm*)↑.*nRightArgs*↑.*Count*;
  lFormatNr ← *gFormatsColl.LookUp_PredFormat*(*nSymbolNr*, *lLeftArgsNbr*, *TermNbr* − *nRightArgBase*);
  if *lFormatNr* = 0 **then** *Error*(*nSubexpPos*, 153); { missing format }
  lRightArgs ← *CreateArgs*(*nRightArgBase* + 1);
  gLastFormula ← *new*(*RightSideOfPredicativeFormulaPtr*, *Init*(*nSubexpPos*, *nSymbolNr*, *lRightArgs*));
  nMultiPredicateList.Insert(*gLastFormula*);
  end;

1464. When the Parser is parsing an atomic formula, when it has parsed a formula and encounters another predicate, it defaults to thinking that it is starting a "multi-predicative formula" (§1540), and it invokes this method. This initializes the *nMultiPredicateList* to an empty list of length 4, and the first entry points to the same formula pointed to by the *gLastFormula* state variable.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.StartMultiPredicativeFormula*;
  begin *nMultiPredicateList.Init*(4); *nMultiPredicateList.Insert*(*gLastFormula*);
  end;

1465. Finishing a "multi-predicative formula" allocates a new *MultiPredicativeFormula* object, and moves the contents of the caller's *nMultiPredicateList* to the newly minted formula. The *gLastFormula* state variable is updated to point to this newly allocated formula object.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishMultiPredicativeFormula*;
  begin *gLastFormula* ← *new*(*MultiPredicativeFormulaPtr*, *Init*(*nSubexpPos*, *new*(*PList*,
    MoveList(*nMultiPredicateList*))));
  end;

1466. The Parser has just parsed "⟨*Term*⟩ `is` ⟨*Type*⟩", 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 "⟨*Term*⟩ `is not` ⟨*Type*⟩", 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).

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishQualifyingFormula*;
  var *j*: *integer*;
  begin *mizassert*(5430, *gLastType* ≠ **nil**);
  if *nAttrCollection.Count* > 0 **then**
    begin *gLastType* ← *new*(*ClusteredTypePtr*, *Init*(*gLastType*↑.*nTypePos*, *new*(*PList*,
      Init(*nAttrCollection.Count*)), *gLastType*));
    for *j* ← 0 **to** *nAttrCollection.Count* − 1 **do**
      ClusteredTypePtr(*gLastType*)↑.*nAdjectiveCluster*↑.*Insert*(*PObject*(*nAttrCollection.Items*↑[*j*]));
    end;
  gLastFormula ← *new*(*QualifyingFormulaPtr*, *Init*(*nSubexpPos*, *Term*[*TermNbr*], *gLastType*));
  if *nPostNegated* **then** *gLastFormula* ← *new*(*NegativeFormulaPtr*, *Init*(*nNotPos*, *gLastFormula*));
  dec(*TermNbr*);
  end;

1467. The Parser has just finished parsing "⟨*Term*⟩ `is` ⟨*Attribute*⟩" or "⟨*Term*⟩ `is not` ⟨*Attribute*⟩", 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).

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishAttributiveFormula*;
  begin *gLastFormula* ← *new*(*AttributiveFormulaPtr*, *Init*(*nSubExpPos*, *Term*[*TermNbr*], *new*(*PList*,
    MoveList(*nAttrCollection*))));
  if *nPostNegated* **then** *gLastFormula* ← *new*(*NegativeFormulaPtr*, *Init*(*nNotPos*, *gLastFormula*));
  dec(*TermNbr*);
  end;

1468. While the Parser is working its way through a formula, and it is looking at an identifier and the next token is a square bracket "`[`", then the Parser invokes this method to initialize the relevant fields to store accumulated data.

⟨Extended subexpression implementation 1390⟩ +≡
procedure *extSubexpObj.StartPrivateFormula*;
  begin *nTermBase* ← *TermNbr*; *nSubexpPos* ← *CurPos*; *nSpelling* ← *CurWord.Nr*;
  end;

1469. The Parser has just encountered "`]`" and now we assemble the accumulated data into a formula. This allocates a new *PrivatePredicativeFormula*, moves the arguments encountered since starting the private predicate into a list (§1393) owned by the formula object. The *gLastFormula* is updated to point to the newly allocated formula object.

⟨Extended subexpression implementation 1390⟩ +≡
procedure *extSubexpObj.FinishPrivateFormula*;
  begin *gLastFormula* ← *new*(*PrivatePredicativeFormulaPtr*, *Init*(*nSubexpPos*, *nSpelling*,
      CreateArgs(*nTermBase* + 1)));
  end;

1470. The Parser has encountered the "`contradiction`" token, so it invokes this method, which allocates a *ContradictionFormula* and updates the *gLastFormula* state variable to point to it.

⟨Extended subexpression implementation 1390⟩ +≡
procedure *extSubexpObj.ProcessContradiction*;
  begin *gLastFormula* ← *new*(*ContradictionFormulaPtr*, *Init*(*CurPos*));
  end;

1471. The Parser routinely allocates a formula object, then realizes later it should negate that formula object. This is handled by storing the formula object in the *gLastFormula* object, then this method allocates a new formula (which is the negation of the *gLastFormula*) and updates the *gLastFormula* to point to the newly allocated negated formula.

⟨Extended subexpression implementation 1390⟩ +≡
procedure *extSubexpObj.ProcessNegative*;
  begin *gLastFormula* ← *new*(*NegativeFormulaPtr*, *Init*(*CurPos*, *gLastFormula*));
  end;

1472. When the Parser has encountered the "`not`" reserved keyword, it invokes the *ProcessNegation* method which just toggles the *nPostNegated* field of the caller, and assigns the *nNotPos* field to the Parser's current position.

⟨Extended subexpression implementation 1390⟩ +≡
procedure *extSubexpObj.ProcessNegation*;
  begin *nPostNegated* ← ¬*nPostNegated*; *nNotPos* ← *CurPos*;
  end;

1473. When the Parser is looking at a binary connective token (e.g., "`implies`", "`or`", etc.), this method is invoked to store the connective kind as well as the "left-hand side" to the binary connective in the *nFirstSententialOperand* field.

⟨Extended subexpression implementation 1390⟩ +≡
procedure *extSubexpObj.ProcessBinaryConnective*;
  begin *nConnective* ← *CurWord.Kind*; *nFirstSententialOperand* ← *gLastFormula*;
  nSubexpPos ← *CurPos*;
  end;

1474. The Parser has seen "⟨*Formula*⟩ `or ... or`". Then this method will be invoked to store that first formula parsed in the caller's *nFirstSententialOperand* field.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.ProcessFlexDisjunction*;
  begin *nFirstSententialOperand* ← *gLastFormula*;
  end;

1475. The Parser has seen "⟨*Formula*⟩ `& ... &`". Then this method will be invoked to store that first formula parsed in the caller's *nFirstSententialOperand* field.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.ProcessFlexConjunction*;
  begin *nFirstSententialOperand* ← *gLastFormula*;
  end;

1476. The Parser has parsed "`for` ⟨*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.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.StartRestriction*;
  begin *nRestrPos* ← *CurPos*;
  end;

1477. The Parser has just parsed the formula appearing after "`st`", so this method is invoked to store that formula in the caller's *nRestriction* field (for later use when constructing an AST).

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishRestriction*;
  begin *nRestriction* ← *gLastFormula*;
  end;

1478. 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.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishBinaryFormula*;
  begin case *nConnective* **of**
  sy_Implies: *gLastFormula* ← *new*(*ConditionalFormulaPtr*, *Init*(*nSubExpPos*, *nFirstSententialOperand*, *gLastFormula*));
  sy_Iff: *gLastFormula* ← *new*(*BiconditionalFormulaPtr*, *Init*(*nSubexpPos*, *nFirstSententialOperand*, *gLastFormula*));
  sy_Or: *gLastFormula* ← *new*(*DisjunctiveFormulaPtr*, *Init*(*nSubexpPos*, *nFirstSententialOperand*, *gLastFormula*));
  sy_Ampersand: *gLastFormula* ← *new*(*ConjunctiveFormulaPtr*, *Init*(*nSubexpPos*, *nFirstSententialOperand*, *gLastFormula*));
  othercases *RunTimeError*(3124);
  endcases;
  end;

1479. We have parsed "⟨*Formula*⟩ `or ... or` ⟨*Formula*⟩", 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 1390 ⟩ +≡
procedure *extSubexpObj.FinishFlexDisjunction*; { polaczyc z flexConj }
begin *gLastFormula* ← *new*(*FlexaryDisjunctiveFormulaPtr*, *Init*(*CurPos*, *nFirstSententialOperand*, *gLastFormula*));
end;

1480. We have parsed "⟨*Formula*⟩ `& ... &` ⟨*Formula*⟩", 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.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishFlexConjunction*;
begin *gLastFormula* ← *new*(*FlexaryConjunctiveFormulaPtr*, *Init*(*CurPos*, *nFirstSententialOperand*, *gLastFormula*));
end;

1481. The Parser is looking at the "`ex`" token, then invokes this method to reset the caller's fields in preparation for accumulating data needed when constructing the formula's AST.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.StartExistential*;
begin *nQualifiedSegments.Init*(0); *nSubexpPos* ← *CurPos*;
end;

1482. The Parser is looking at the "`for`" token, and it invokes this method to reset the relevant fields in the caller.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.StartUniversal*;
begin *nQualifiedSegments.Init*(0); *nSubexpPos* ← *CurPos*;
end;

1483. After the Parser has invoked *StartUniversal* or *StartExistential*, it parses the quantified variables (which begins by invoking this method).

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.StartQualifiedSegment*;
begin *nSegmentIdentColl.Init*(2); *nSegmentPos* ← *CurPos*;
end;

1484. The Parser has parsed a comma-separated list and is expecting either "`be`" or "`being`", but before parsing for that copula the Parser invokes the *StartQualifyingType* method to update the *gLastType* state variable to point to **nil**.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.StartQualifyingType*;
begin *gLastType* ← **nil**;
end;

1485. 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* ≠ **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.

⟨Extended subexpression implementation 1390⟩ +≡
```
procedure extSubexpObj.FinishQualifiedSegment;
  var k: integer;
  begin if gLastType = nil then
    begin for k ← 0 to nSegmentIdentColl.Count − 1 do
      begin nQualifiedSegments.Insert(new(ImplicitlyQualifiedSegmentPtr,
          Init(VariablePtr(nSegmentIdentColl.Items↑[k])↑.nVarPos, nSegmentIdentColl.Items↑[k])));
      nSegmentIdentColl.Items↑[k] ← nil;
      end;
    nSegmentIdentColl.Done;
    end
  else begin nQualifiedSegments.Insert(new(ExplicitlyQualifiedSegmentPtr, Init(nSegmentPos,
        new(PList, MoveList(nSegmentIdentColl)), gLastType)));
    end;
  end;
```

1486. When the Parser is parsing quantified variables, specifically when it is parsing a comma-separated list of variables, it will invoke this method, then check if the next token is a comma (and if so iterate). This *ProcessVariable* method should accumulate a *Variable* object with the current token's identifier, then insert it into the caller's *nSegmentIdentColl* field.

⟨Extended subexpression implementation 1390⟩ +≡
```
procedure extSubexpObj.ProcessVariable;
  begin nSegmentIdentColl.Insert(new(VariablePtr, Init(CurPos, GetIdentifier)));
  end;
```

1487. The Parser has just finished something like

$$\texttt{ex}\ \langle\mathit{Qualified\text{-}Variables}\rangle\ \texttt{,}\ \ldots\ \texttt{,}\ \langle\mathit{Qualified\text{-}Variables}\rangle\ \texttt{st}\ \langle\mathit{Formula}\rangle$$

Now we assemble it as

$$\texttt{ex}\ \langle\mathit{Qualified\text{-}Variables}\rangle\ \texttt{st}\ \bigl(\texttt{ex}\ \ldots\ \texttt{st}\ (\texttt{ex}\ \langle\mathit{Qualified\text{-}Variables}\rangle\ \texttt{st}\ \langle\mathit{Formula}\rangle)\bigr)$$

starting with the innermost existentially quantified formula, working our ways outwards.

Importantly, assembling the AST reflects the quantified variables has the grammar

$$\begin{aligned}\langle\mathit{Qualified\text{-}Variables}\rangle &= \langle\mathit{Implicitly\text{-}Qualified\text{-}Variables}\rangle\\ &\quad\mid \langle\mathit{Explicitly\text{-}Qualified\text{-}Variables}\rangle\\ &\quad\mid \langle\mathit{Explicitly\text{-}Qualified\text{-}Variables}\rangle\ \texttt{","}\ \langle\mathit{Implicitly\text{-}Qualified\text{-}Variables}\rangle\end{aligned}$$

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishExistential*;
  var *k*: *integer*;
  begin for $k \leftarrow nQualifiedSegments.Count - 1$ **downto** 1 **do** { from inside outwards }
    begin *gLastFormula* ← *new*(*ExistentialFormulaPtr*,
        Init(*QualifiedSegmentPtr*(*nQualifiedSegments.Items*↑[*k*])↑.*nSegmPos*,
        nQualifiedSegments.Items↑[*k*], *gLastFormula*)); *nQualifiedSegments.Items*↑[*k*] ← **nil**;
    end;
  if *nQualifiedSegments.Count* > 0 **then**
    begin *gLastFormula* ← *new*(*ExistentialFormulaPtr*, *Init*(*nSubexpPos*, *nQualifiedSegments.Items*↑[0],
        gLastFormula)); *nQualifiedSegments.Items*↑[0] ← **nil**;
    end;
  nQualifiedSegments.Done;
  end;

1488. Universally quantified formulas first transforms

$$\texttt{for}\ \langle\mathit{Qualified\text{-}Variables}\rangle\ \texttt{st}\ \langle\mathit{Formula}\rangle_1\ \texttt{holds}\ \langle\mathit{Formula}\rangle_2$$

into

$$\texttt{for}\ \langle\mathit{Qualified\text{-}Variables}\rangle\ \texttt{holds}\ \langle\mathit{Formula}\rangle_1\ \texttt{implies}\ \langle\mathit{Formula}\rangle_2$$

which is handled immediately.

The remainder of the method iteratively constructs the universally quantified formulas by "unrolling" the qualified segments, just as we did for existentially quantified formulas.

⟨ Extended subexpression implementation 1390 ⟩ +≡
procedure *extSubexpObj.FinishUniversal*;
  var *k*: *integer*;
  begin if *nRestriction* ≠ **nil then** { transform `st` into `implies` }
    gLastFormula ← *new*(*ConditionalFormulaPtr*, *Init*(*nRestrPos*, *nRestriction*, *gLastFormula*));
  for $k \leftarrow nQualifiedSegments.Count - 1$ **downto** 1 **do**
    begin *gLastFormula* ← *new*(*UniversalFormulaPtr*,
        Init(*QualifiedSegmentPtr*(*nQualifiedSegments.Items*↑[*k*])↑.*nSegmPos*,
        nQualifiedSegments.Items↑[*k*], *gLastFormula*)); *nQualifiedSegments.Items*↑[*k*] ← **nil**;
    end;
  if *nQualifiedSegments.Count* > 0 **then**
    begin *gLastFormula* ← *new*(*UniversalFormulaPtr*, *Init*(*nSubexpPos*, *nQualifiedSegments.Items*↑[0],
        gLastFormula)); *nQualifiedSegments.Items*↑[0] ← **nil**;
    end;
  end;

Section 23.4. EXTENDED EXPRESSION CLASS

1489. When an expression is needed, the *gExpPtr* state variable is used to build it out of subexpressions. The *gExpPtr* state variable is an instance of the *extExpression* class.

⟨ Extended expression class declaration 1489 ⟩ ≡
  extExpressionPtr = ↑*extExpressionObj*;
  extExpressionObj = **object** (*ExpressionObj*)
    constructor *Init*(*fExpKind* : *ExpKind*);
    procedure *CreateSubexpression*; *virtual*;
  end ;

This code is used in section 1199.

1490. Constructor. This just invokes the parent class's constructor (§711), then resets the module-wide variable *TermNbr* to zero.

⟨ Extended expression implementation 1490 ⟩ ≡
constructor *extExpressionObj*.*Init*(*fExpKind* : *ExpKind*);
  begin *inherited* *Init*(*fExpKind*); *TermNbr* ← 0;
  end;

See also section 1491.

This code is used in section 1200.

1491. An *extExpression* creating a subexpression *overrides* the parent class's method (§712), and sets the global *gSubexpPtr* to point to a new *extSubexp* object.

⟨ Extended expression implementation 1490 ⟩ +≡
procedure *extExpressionObj*.*CreateSubexpression*;
  begin *gSubexpPtr* ← *new*(*extSubexpPtr*, *Init*)
  end;

File 24

Parser

1492. The parser has a "big red button": a single "obvious" function for the user to, you know, push. Namely, the *Parse* procedure (§§1658 *et seq.*). Everything else is just a helper function.

The design of the parser appears to be a recursive descent parser on statements, with parsing expressions handled specially.

Note that the `base/parser.pas` file appears to be naturally divided up into sections, with comments which appear to use the Germanic "s p a c i n g f o r i t a l i c s" (which I have just replaced with more readable *italicized* versions). I have used these cleavages to organize the discussion of this file.

The *StillCorrect* global variable is *false* when the parser has entered what programmers call **"Panic Mode"**: something has gone awry, and the parser is trying to recover gracefully. For a friendly review of panicking, see Bob Nystrom's *Crafting Interpreters* (Chaper 6, Section 3).

⟨ parser.pas 1492 ⟩ ≡
  ⟨ GNU License 4 ⟩
unit *parser*;
  interface
  uses *mscanner*;
  var *StillCorrect*: *boolean* = *true*;
  type *ReadTokenProcedure* = *Procedure*;
  const *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 1493 ⟩

1493. We have a few constants, but the implementation is loosely organized around parsing expressions (terms and formulas), statements, and then blocks.

⟨ Implementation of parser.pas 1493 ⟩ ≡
  ⟨ Local constants for parser.pas 1494 ⟩;
  ⟨ Parse expressions (`parser.pas`) 1502 ⟩
  ⟨ Communicate with items (`parser.pas`) 1558 ⟩
  ⟨ Process miscellany (`parser.pas`) 1559 ⟩
  ⟨ Parse simple justifications (`parser.pas`) 1571 ⟩
  ⟨ Parse statements and reasoning (`parser.pas`) 1577 ⟩
  ⟨ Parse patterns (`parser.pas`) 1602 ⟩
  ⟨ Parse definitions (`parser.pas`) 1621 ⟩
  ⟨ Parse scheme block (`parser.pas`) 1654 ⟩
  ⟨ Main parse method (`parser.pas`) 1658 ⟩

See also sections 1495, 1496, 1498, 1499, 1500, and 1501.

This code is used in section 1492.

1494. We have error codes for syntactically invalid situations. These are all different ways for panic to occur (hence the "pa-" prefix).

⟨ Local constants for parser.pas 1494 ⟩ ≡
const *paUnexpOf* = 183; *paUnexpOver* = 184; *paUnexpEquals* = 186; *paUnexpAntonym1* = 198;
paUnexpAntonym2 = 198; *paUnexpSynonym* = 199; *paUnpairedSymbol* = 214; *paEndExp* = 215;
paUnexpHereby = 216; *paAdjClusterExp* = 223; *paUnexpReconsider* = 228; *paPerExp* = 231;
paSupposeOrCaseExp = 232; *paOfExp* = 256; *paUnexpRedef* = 273; *paAllExp* = 275;
paIdentExp1 = 300; *paIdentExp2* = 300; *paIdentExp3* = 300; *paIdentExp4* = 300; *paIdentExp5* = 300;
paIdentExp6 = 300; *paIdentExp7* = 300; *paIdentExp8* = 300; *paIdentExp9* = 300; *paIdentExp10* = 300;
paIdentExp11 = 300; *paIdentExp12* = 300; *paIdentExp13* = 300; *paWrongPredPattern* = 301;
paFunctExp1 = 302; *paFunctExp2* = 302; *paFunctExp3* = 302; *paFunctExp4* = 302;
paWrongModePatternBeg = 303; *paStructExp1* = 304; *paSelectExp1* = 305; *paAttrExp1* = 306;
paAttrExp2 = 306; *paAttrExp3* = 306; *paNumExp* = 307; *paWrongReferenceBeg* = 308;
paTypeOrAttrExp = 309; *paRightBraExp1* = 310; *paRightBraExp2* = 310;
paWrongRightBracket1 = 311; *paWrongRightBracket2* = 311; *paDefExp* = 312; *paSchExp* = 313;
paWrongPattBeg1 = 314; *paWrongPattBeg2* = 314; *paWrongPattBeg3* = 314;
paWrongModePatternSet = 315; *paWrongAfterThe* = 320; *paWrongPredSymbol* = 321;
paSemicolonExp = 330; *paUnexpConnective* = 336; *paWrongScopeBeg* = 340; *paThatExp1* = 350;
paThatExp2 = 350; *paCasesExp* = 351; *paLeftParenthExp* = 360; *paLeftSquareExp* = 361;
paLeftCurledExp = 362; *paLeftDoubleExp1* = 363; *paLeftDoubleExp3* = 363;
paWrongSchemeVarQual = 364; *paRightParenthExp1* = 370; *paRightParenthExp2* = 370;
paRightParenthExp3 = 370; *paRightParenthExp4* = 370; *paRightParenthExp5* = 370;
paRightParenthExp6 = 370; *paRightParenthExp7* = 370; { forgot right paren in "`from SCHEME(`" }
paRightParenthExp8 = 370; *paRightParenthExp9* = 370; *paRightParenthExp10* = 370;
paRightParenthExp11 = 370; *paRightSquareExp1* = 371; *paRightSquareExp2* = 371;
paRightSquareExp3 = 371; *paRightSquareExp4* = 371; *paRightSquareExp5* = 371;
paRightCurledExp1 = 372; *paRightCurledExp2* = 372; *paRightCurledExp3* = 372;
paRightDoubleExp1 = 373; *paRightDoubleExp2* = 373; *paWrongAttrPrefixExpr* = 375;
paWrongAttrArgumentSuffix = 376; *paTypeExpInAdjectiveCluster* = 377; *paEqualityExp1* = 380;
paEqualityExp2 = 380; *paIfExp* = 381; *paForExp* = 382; *paIsExp* = 383; *paColonExp1* = 384;
paColonExp2 = 384; *paColonExp3* = 384; *paColonExp4* = 384; *paArrowExp1* = 385;
paArrowExp2 = 385; *paMeansExp* = 386; *paStExp* = 387; *paAsExp* = 388; *paProofExp* = 389;
paWithExp = 390; *paWrongItemBeg* = 391; *paUnexpItemBeg* = 392; *paWrongJustificationBeg* = 395;
paWrongFormulaBeg = 396; *paWrongTermBeg* = 397; *paWrongRadTypeBeg* = 398;
paWrongFunctorPatternBeg = 399; *paStillNotImplemented* = 400; *paNotExpected* = 401;
paInfinitiveExp = 402; *paSuchExp* = 403; *paToExp* = 404; *paTypeUnexpInClusterRegistration* = 405;
paForOrArrowExpected = 406;

See also section 1497.

This code is used in section 1493.

1495. ⟨ Implementation of parser.pas 1493 ⟩ +≡
var *gAddSymbolsSet*: **set of** *char* = [ ]; { not used anywhere }

1496. Syntax errors do three things:
(1) Marks *StillCorrect* to be false (i.e., enters panic mode)
(2) Reports the error with the *ErrImm* (§106) function.
(3) Skips ahead until we find a token in the *gMainSet*, then try to proceed like things are still alright (so we "fail gracefully").

⟨ Implementation of parser.pas 1493 ⟩ +≡
procedure *SynErr*(*fPos* : *Position*; *fErrNr* : *integer*);
  begin if *StillCorrect* **then**
    begin *StillCorrect* ← *false*;
    if *CurWord.Kind* = *sy_Error* **then**
      begin if *CurWord.Nr* ≠ *scTooLongLineErrorNr* **then** *ErrImm*(*CurWord.Nr*)
      else *Error*(*fPos*, *fErrNr*);
      end
    else *Error*(*fPos*, *fErrNr*);
    while ¬(*CurWord.Kind* ∈ *gMainSet*) **do** *ReadTokenProc*;
    end;
  end;

1497. 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.

⟨ Local constants for parser.pas 1494 ⟩ +≡
const *gMainSet*: **set of** *TokenKind* = [*sy_Begin*, *sy_Semicolon*, *sy_Proof*, *sy_Now*, *sy_Hereby*, *sy_Definition*, *sy_End*, *sy_Theorem*, *sy_Reserve*, *sy_Notation*, *sy_Registration*, *sy_Scheme*, *EOT*, *sy_Deffunc*, *sy_Defpred*, *sy_Reconsider*, *sy_Consider*, *sy_Then*, *sy_Per*, *sy_Case*, *sy_Suppose*];

1498. We have a few more methods for *specific* kinds of errors we are likely to encounter.

⟨ Implementation of parser.pas 1493 ⟩ +≡
procedure *MissingWord*(*fErrNr* : *integer*);
  var *lPos*: *Position*;
  begin *lPos* ← *PrevPos*; *inc*(*lPos.Col*); *SynErr*(*lPos*, *fErrNr*)
  end;

procedure *WrongWord*(*fErrNr* : *integer*);
  begin *SynErr*(*CurPos*, *fErrNr*)
  end;

1499. We will want to assert the parser has encountered a specific token (like a semicolon or "`end`") and raise an error if it has not. This will make for much more readable code later on. We should recall *KillItem* (§686) mutates the global state.

⟨ Implementation of parser.pas 1493 ⟩ +≡
procedure *Semicolon*;
  begin *KillItem*;
  if *CurWord.Kind* ≠ *sy_Semicolon* **then** *MissingWord*(*paSemicolonExp*);
  if *CurWord.Kind* = *sy_Semicolon* **then** *ReadTokenProc*;
  end;

procedure *AcceptEnd*(*fPos* : *Position*);
  begin if *CurWord.Kind* = *sy_End* **then** *ReadTokenProc*
  else begin *Error*(*fPos*, *paEndExp*); *MissingWord*(*paUnpairedSymbol*)
    end;
  end;

1500. These previous methods can be generalized to an *Accept* procedure which checks whether a given *TokenKind* has "occurred". If so, just read the next word. Otherwise, flag an error.

⟨ Implementation of parser.pas 1493 ⟩ +≡
procedure *ReadWord*;
  begin *Mizassert*(2546, *StillCorrect*); *ReadTokenProc*
  end;

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;

1501. Flagging a semantic error should first check if we are in "panic mode" or not. If we are already panicking, there's no reason to heap more panicky error messages onto the screen.

⟨ Implementation of parser.pas 1493 ⟩ +≡
procedure *SemErr*(*fErrNr* : *integer*);
  begin if *StillCorrect* **then** *ErrImm*(*fErrNr*)
  end;

Section 24.1. EXPRESSIONS

1502. We have a few token kinds which indicate the start of a term:
(1) identifiers (for variables and private functors),
(2) infixed operators,
(3) numerals,
(4) left and right brackets of all sorts,
(5) the anaphoric "`it`" constant used in definitions,
(6) "`the`" choice operator,
(7) placeholder variables appearing in private functors and predicates,
(8) structure symbols.

⟨ Parse expressions (`parser.pas`) 1502 ⟩ ≡
{ *Expressions* }
const *TermBegSys*: **set of**
TokenKind = [*Identifier*, *InfixOperatorSymbol*, *Numeral*, *LeftCircumfixSymbol*, *sy_LeftParanthesis*, *sy_It*, *sy_LeftCurlyBracket*, *sy_LeftSquareBracket*, *sy_The*, *sy_Dolar*, *Structuresymbol*];

See also sections 1503, 1504, 1505, 1506, 1508, 1518, 1519, 1520, 1523, 1525, 1530, 1531, 1533, 1534, 1535, 1537, 1538, 1539, 1540, 1543, 1544, 1549, 1552, 1553, 1554, 1555, 1556, and 1557.

This code is used in section 1493.

Subsection 24.1.1. Terms

1503. We have a few helper function for *Accept*-ing parentheses. This invokes the *ProcessLeftParenthesis* method for the *gSubexpPtr* (§690) global variable which we recall (§718) is an empty virtual method. So the parser just "consumes" a left parentheses, and will continue to read tokens while they are left parentheses. The argument passed in will be mutated to track the number of left parentheses consumed.

Similarly, the *CloseParenth* method will have the compiler consume right parentheses, mutating the argument passed in (to decrement the number of right parentheses consumed). This will let us track mismatched parentheses errors.

⟨ Parse expressions (`parser.pas`) 1502 ⟩ +≡

```
procedure OpenParenth(var fParenthCnt : integer);
  begin fParenthCnt ← 0;
  while CurWord.Kind = sy_LeftParanthesis do
    begin gSubexpPtr↑.ProcessLeftParenthesis; ReadWord; inc(fParenthCnt);
    end;
  end;
procedure CloseParenth(var fParenthCnt : integer);
  begin while (CurWord.Kind = sy_RightParanthesis) ∧ (fParenthCnt > 0) do
    begin dec(fParenthCnt); gSubexpPtr↑.ProcessRightParenthesis; ReadWord;
    end;
  end;
```

1504. Qualified expressions. Parsing qualified expressions includes a control flow for "exactly" qualified expressions.

We should recall from "Mizar in a nutshell" that the "`exactly`" keyword is reserved but not currently used for anything. The global subexpression pointer is invoking empty virtual methods (§718). So what's going on?

Well, the only work being done here is in the branch handling "`qua`", specifically the next word is read, and then control is handed off to *TypeSubexpression*.

⟨Parse expressions (`parser.pas`) 1502⟩ +≡
procedure *TypeSubexpression*; *forward*;

procedure *AppendQua*;
  begin while *CurWord.Kind* = *sy_Qua* **do**
    begin *gSubexpPtr*↑.*ProcessQua*; *ReadWord*; *TypeSubexpression*; *gSubexpPtr*↑.*FinishQualifiedTerm*;
    end;
  if *CurWord.Kind* = *sy_Exactly* **then**
    begin *gSubexpPtr*↑.*ProcessExactly*; *ReadWord*
    end;
  end;

1505. Parsing *the contents of* a bracketed term starts a bracketed term (§718), reads the next word after the start of the bracket, then consumes the maximum number of visible arguments (§682).

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.

⟨Parse expressions (`parser.pas`) 1502⟩ +≡
procedure *GetArguments*(**const** *fArgsNbr*: *integer*); *forward*;

procedure *BracketedTerm*;
  begin *gSubexpPtr*↑.*StartBracketedTerm*; *ReadWord*; *GetArguments*(*MaxVisArgNbr*);
  gSubexpPtr↑.*FinishBracketedTerm*;
  end;

1506. Parsing post-qualified variables (i.e., variables which appear in a Fraenkel term's "`where`" clause) which consists of a comma-separated list of post-qualified segments.

⟨Parse expressions (`parser.pas`) 1502⟩ +≡
procedure *TermSubexpression*; *forward*;
procedure *FormulaSubexpression*; *forward*;
procedure *ArgumentsTail*(*fArgsNbr* : *integer*); *forward*;

procedure *ProcessPostqualification*;
  begin *gSubexpPtr*↑.*StartPostqualification*;
  while *CurWord.Kind* = *sy_Where* **do**
    begin repeat ⟨Process post-qualified segment 1507⟩
    until *CurWord.Kind* ≠ *sy_Comma*;
    end;
  end;

1507. Each "segment" in a post-qualification looks like:

$$\langle variable\rangle\ \{\texttt{","}\ \langle variable\rangle\}\ (\texttt{"is"}\mid\texttt{"being"})\ \langle type\rangle$$

We can process the comma-separated list of variables, then the type ascription term ("is" or "being"), then process the type.

define *process_postqualified_variables* ≡ **repeat** *gSubexpPtr*↑.*ProcessPostqualifiedVariable*;
Accept(*Identifier*, *paIdentExp1*);
until ¬*Occurs*(*sy_Comma*)
define *post_qualified_type* ≡ **begin** *ReadWord*; *TypeSubexpression*; **end**

⟨ Process post-qualified segment 1507 ⟩ ≡
gSubexpPtr↑.*StartPostQualifyingSegment*; *ReadWord*;
process_postqualified_variables;
gSubexpPtr↑.*StartPostqualificationSpecyfication*;
if *CurWord*.*Kind* ∈ [*sy_Is*, *sy_are*] **then** *post_qualified_type*;
gSubexpPtr↑.*FinishPostqualifyingSegment*;

This code is used in section 1506.

1508. Getting a closed subterm is part of the loop for parsing a term. The intricate relationship of mutually recursive function calls looks something like the following (assuming there are no parsing errors):

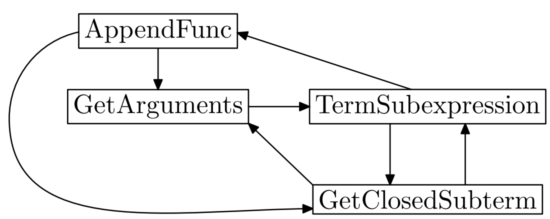

Fig. 8. Control flow when parsing a term.

The *GetArguments* parses a comma-separated list of terms. Since each term in the comma-separated list will be a *subterm* of a larger expression, we parse it with *TermSubexpression* (which invokes *GetClosedSubterm* in a mutually recursive relation). If there is a chain of infix operators (like $x + y - z \times \omega$), then *AppendFunc* is invoked on the infixed operators.

⟨ Parse expressions (`parser.pas`) 1502 ⟩ +≡
procedure *GetClosedSubterm*;
begin case *CurWord*.*Kind* **of**
⟨ Get closed subterm of identifier 1509 ⟩;
⟨ Get closed subterm of structure 1510 ⟩;
Numeral: **begin** *gSubexpPtr*↑.*ProcessNumeralTerm*; *ReadWord* **end**;
⟨ Get closed subterm of bracketed expression 1511 ⟩;
sy_It: **begin** *gSubexpPtr*↑.*ProcessItTerm*; *ReadWord* **end**;
sy_Dolar: **begin** *gSubexpPtr*↑.*ProcessLocusTerm*; *ReadWord* **end**;
⟨ Get closed subterm of Fraenkel operator or enumerated set 1512 ⟩;
⟨ Get closed subterm of choice operator 1515 ⟩;
othercases *RunTimeError*(2133);
endcases;
end;

1509. If we treat an identifier as a term, then it is either a private functor or it is a variable. How do we tell the difference? A private functor starts with an identifier followed by a left parentheses.

⟨ Get closed subterm of identifier 1509 ⟩ ≡
Identifier: **if** *AheadWord.Kind* = *sy_LeftParanthesis* **then** { treat identifier as private functor }
  begin *gSubexpPtr*↑.*StartPrivateTerm*; *ReadWord*; *ReadWord*;
  if *CurWord.Kind* ≠ *sy_RightParanthesis* **then** *GetArguments*(*MaxVisArgNbr*);
  gSubexpPtr↑.*FinishPrivateTerm*; *Accept*(*sy_RightParanthesis*, *paRightParenthExp2*);
  end
else { treat identifier as variable }
begin *gSubexpPtr*↑.*ProcessSimpleTerm*; *ReadWord* **end**

This code is used in section 1508.

1510. If the parser stumbles across the name of a structure when expecting a term, then the parser should treat it as constructing a new instance of the structure.

⟨ Get closed subterm of structure 1510 ⟩ ≡
StructureSymbol: **begin** *gSubexpPtr*↑.*StartAggregateTerm*; *ReadWord*;
Accept(*sy_StructLeftBracket*, *paLeftDoubleExp1*); *GetArguments*(*MaxVisArgNbr*);
gSubexpPtr↑.*FinishAggregateTerm*; *Accept*(*sy_StructRightBracket*, *paRightDoubleExp1*);
end

This code is used in section 1508.

1511. Encountering a left bracket of some kind should cause the parser to look for the contents of a bracketed term (§1505), then a right bracket.

⟨ Get closed subterm of bracketed expression 1511 ⟩ ≡
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 1508.

1512. When the parser runs into a left curly bracket "{", we either have encountered a Fraenkel operator *or* we have encountered a finite set.

⟨ Get closed subterm of Fraenkel operator or enumerated set 1512 ⟩ ≡
sy_LeftCurlyBracket: **begin** *gSubexpPtr*↑.*StartBracketedTerm*; *ReadWord*; *TermSubexpression*;
if (*CurWord.Kind* = *sy_Colon*) ∨ (*CurWord.Kind* = *sy_Where*) **then** ⟨ Parse a Fraenkel operator 1513 ⟩
else ⟨ Parse an enumerated set 1514 ⟩;
end

This code is used in section 1508.

1513. Parsing a Fraenkel operator, well, we recall Fraenkel operators look like

$$\{\langle\textit{term}\rangle\langle\textit{post-qualified segment}\rangle\ \texttt{":"}\ \langle\textit{formula}\rangle\}$$

⟨ Parse a Fraenkel operator 1513 ⟩ ≡
begin *gSubexpPtr*↑.*StartFraenkelTerm*; *ProcessPostqualification*; *gSubexpPtr*↑.*FinishSample*;
Accept(*sy_Colon*, *paColonExp1*); *FormulaSubexpression*; *gSubexpPtr*↑.*FinishFraenkelTerm*;
Accept(*sy_RightCurlyBracket*, *paRightCurledExp1*);
end

This code is used in section 1512.

1514. We can also run into a finite set $\{x_1, \ldots, x_n\}$.

⟨ Parse an enumerated set 1514 ⟩ ≡
  begin *gSubexpPtr*↑.*FinishArgument*; *ArgumentsTail*(*MaxVisArgNbr* − 1);
  gSubexpPtr↑.*FinishBracketedTerm*;
  case *Curword*.*Kind* **of**
  sy_RightSquareBracket, *sy_RightCurlyBracket*, *sy_RightParanthesis*: *ReadWord*;
  othercases *Accept*(*RightCircumfixSymbol*, *paRightBraExp1*);
  endcases;
  end

This code is used in section 1512.

1515. Mizar allows "`the`" to be used for selector functors, forgetful functors, choice operators, or simple Fraenkel terms.

⟨ Get closed subterm of choice operator 1515 ⟩ ≡
sy_The: **begin** *gSubexpPtr*↑.*ProcessThe*; *ReadWord*;
  case *CurWord*.*Kind* **of**
  SelectorSymbol: **begin** *gSubexpPtr*↑.*StartSelectorTerm*; *ReadWord*;
    if *Occurs*(*sy_Of*) **then** *TermSubexpression*;
    gSubexpPtr↑.*FinishSelectorTerm*;
    end;
  StructureSymbol: ⟨ Parse forgetful functor or choice of structure type 1516 ⟩;
  sy_Set: ⟨ Parse simple Fraenkel expression or "the set" 1517 ⟩;
  ModeSymbol, *AttributeSymbol*, *sy_Non*, *sy_LeftParanthesis*, *Identifier*, *InfixOperatorSymbol*, *Numeral*,
      LeftCircumfixSymbol, *sy_It*, *sy_LeftCurlyBracket*, *sy_LeftSquareBracket*, *sy_The*, *sy_Dolar*:
    begin *gSubexpPtr*↑.*StartChoiceTerm*; *TypeSubexpression*; *gSubexpPtr*↑.*FinishChoiceTerm*;
    end
  othercases begin *gSubexpPtr*↑.*InsertIncorrTerm*; *WrongWord*(*paWrongAfterThe*) **end**;
  endcases;
  end

This code is used in section 1508.

1516. A forgetful functor always looks like

`"the"` ⟨*structure*⟩ `"of"` ⟨*term*⟩

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"`.

⟨ Parse forgetful functor or choice of structure type 1516 ⟩ ≡
  if *AheadWord*.*Kind* = *sy_Of* **then** { forgetful functor }
    begin *gSubexpPtr*↑.*StartForgetfulTerm*; *ReadWord*; *Accept*(*sy_Of*, *paOfExp*); *TermSubexpression*;
    gSubexpPtr↑.*FinishForgetfulTerm*;
    end
  else { choice operator }
  begin *gSubexpPtr*↑.*StartChoiceTerm*; *TypeSubexpression*; *gSubexpPtr*↑.*FinishChoiceTerm*;
  end

This code is used in section 1515.

1517. Mizar allows "`the set of`" to start a simple Fraenkel expression. But we could also refer to "`the set`" as the set chosen by the axiom of choice.

⟨ Parse simple Fraenkel expression or "the set" 1517 ⟩ ≡
if *AheadWord.Kind* = *sy_Of* **then** { simple Fraenkel expression }
begin *ReadWord*; { set }
ReadWord; { of }
gSubexpPtr↑.*StartSimpleFraenkelTerm*; *Accept*(*sy_All*, *paAllExp*); *TermSubexpression*;
gSubexpPtr↑.*StartFraenkelTerm*; *ProcessPostqualification*; *gSubexpPtr*↑.*FinishSimpleFraenkelTerm*;
end
else { "the set" }
begin *gSubexpPtr*↑.*StartChoiceTerm*; *TypeSubexpression*; *gSubexpPtr*↑.*FinishChoiceTerm*; **end**

This code is used in section 1515.

1518. Subexpression object's *FinishArgument* (§1405) is invoked. This will invoke the *AppendQua* (§1504) method and expect a closed parentheses afterwards (§1503).

Possible bug: what should happen when *fParenthCnt* is zero or negative?

⟨ Parse expressions (`parser.pas`) 1502 ⟩ +≡
procedure *CompleteArgument*(**var** *fParenthCnt* : *integer*);
begin *gSubexpPtr*↑.*FinishArgument*;
repeat *AppendQua*; *CloseParenth*(*fParenthCnt*);
until *CurWord.Kind* ≠ *sy_Qua*; { ∧(*CurWord.Kind* ≠ *sy_Exactly*) }
end;

1519. Keep parsing "infixed operators". When the current token is an infixed operator, this will consume the arguments to its right, then iterate. It's also worth remembering that *gExpPtr* (§690) was a global variable declared back in `syntax.pas`, and the *CreateSubexpression* (§1491) mutates the *gSubexpPtr* variable. Now we see it in action.

This invokes the *ProcessLeftParenthesis* method for the *gSubexpPtr* (§690) global variable which we recall (§718) is an empty virtual method. So the parser just "consumes" a left parentheses.

Note that the **case** expression considers the type of *TokenKind* (§724) of the current word. But it is not exhaustive.

There is a comment in Polish, "Chyba po prostu TermSubexpression", which Google translated into English as "I guess it's just Term Subexpression". I swapped this in the code below.

⟨Parse expressions (`parser.pas`) 1502⟩ +≡

```
procedure AppendFunc(var fParenthCnt : integer);
  begin while CurWord.Kind = InfixOperatorSymbol do
    begin gSubexpPtr↑.StartLongTerm;   { (§1413) }
    repeat gSubexpPtr↑.ProcessFunctorSymbol;   { (§1429) }
      ReadWord;
      case CurWord.Kind of
      sy_LeftParanthesis:
        begin   { parenthetised term(s) }
        gSubexpPtr↑.ProcessLeftParenthesis; ReadWord;   { consume the left paren }
        GetArguments(MaxVisArgNbr);   { (§1534) }
        gSubexpPtr↑.ProcessRightParenthesis; Accept(sy_RightParanthesis, 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↑.CreateSubexpression;   { (§1491) }
        GetClosedSubterm;   { (§1508) }
        gSubexpPtr↑.FinishArgument;   { (§1405) }
        KillSubexpression;   { (§684) }
        end;
      endcases;
      gSubexpPtr↑.FinishArgList;   { (§1430) }
    until CurWord.Kind ≠ InfixOperatorSymbol;
    gSubexpPtr↑.FinishLongTerm;   { (§1417) }
    CompleteArgument(fParenthCnt);   { (§1518) }
    end;
  end;
```

1520. Parse terms with infix operators. Note this appears to parse infixed operators as left-associative (e.g., $x + y + z$ is parsed as $(x + y) + z$).

⟨ Parse expressions (`parser.pas`) 1502 ⟩ +≡
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 *gSubexpPtr*↑.*InsertIncorrTerm*; *gSubexpPtr*↑.*FinishArgument*;
    WrongWord(*paWrongTermBeg*);
    end;
  endcases;
  ⟨ Keep parsing as long as there is an infixed operator to the right 1521 ⟩;
  ⟨ Check every remaining open (left) parentheses has a corresponding partner 1522 ⟩;
  end;

1521. ⟨ Keep parsing as long as there is an infixed operator to the right 1521 ⟩ ≡
  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*↑.*ProcessRightParenthesis*; *ReadWord*;
        end;
      end;
  until *CurWord.Kind* $\neq$ *InfixOperatorSymbol*

This code is used in section 1520.

1522. ⟨ Check every remaining open (left) parentheses has a corresponding partner 1522 ⟩ ≡
  while *lParenthCnt* > 0 **do**
    begin *gSubexpPtr*↑.*ProcessRightParenthesis*; *Accept*(*sy_RightParanthesis*, *paRightParenthExp1*);
    dec(*lParenthCnt*);
    end

This code is used in section 1520.

1523. An adjective cluster is just one or more (possibly negated) attribute.

⟨ Parse expressions (`parser.pas`) 1502 ⟩ +≡
  ⟨ Process attributes (`parser.pas`) 1524 ⟩
procedure *GetAdjectiveCluster*;
  begin *gSubexpPtr*↑.*StartAdjectiveCluster*; *ProcessAttributes*; *gSubexpPtr*↑.*FinishAdjectiveCluster*;
  end;

1524. Parsing an attribute amounts to:
(1) handling a leading "`non`"
(2) handling attribute arguments (which always occurs *before* the attribute)
(3) handling the attribute.

define $kind\_is\_radix\_type(\#) \equiv (\# \in [sy\_Set, ModeSymbol, StructureSymbol])$
define $ahead\_is\_attribute\_argument \equiv$
$(CurWord.Kind \in (TermBegSys - [sy\_LeftParanthesis, StructureSymbol])) \lor$
$((CurWord.Kind = sy\_LeftParanthesis) \land \neg(kind\_is\_radix\_type(AheadWord.Kind))) \lor$
$((CurWord.Kind = StructureSymbol) \land (AheadWord.Kind = sy\_StructLeftBracket))$

⟨Process attributes (`parser.pas`) 1524⟩ ≡
procedure *ProcessAttributes*;
begin while $(CurWord.Kind \in [AttributeSymbol, sy\_Non]) \lor ahead\_is\_attribute\_argument$ **do**
begin *gSubexpPtr*↑.*ProcessNon*;
if $CurWord.Kind = sy\_Non$ **then** *ReadWord*;
if *ahead_is_attribute_argument* **then**
begin *gSubexpPtr*↑.*StartAttributeArguments*; *ProcessArguments*;
gSubexpPtr↑.*CompleteAttributeArguments*;
end;
if $CurWord.Kind = AttributeSymbol$ **then**
begin *gSubexpPtr*↑.*ProcessAttribute*; *ReadWord*; **end**
else begin *SynErr*(*CurPos*, *paAttrExp1*) **end**;
end;
end;

This code is used in section 1523.

1525. 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.

⟨Parse expressions (`parser.pas`) 1502⟩ +≡
procedure *RadixTypeSubexpression*;
var *lSymbol*, *lParenthCnt*: *integer*;
begin $lParenthCnt \leftarrow 0$; ⟨Parse optional left-paren 1528⟩;
gSubexpPtr↑.*ProcessModeSymbol*; { (§1408) }
case *CurWord.Kind* **of**
sy_Set: **begin** *ReadWord*;
{ ? if Occurs(syOf) then TypeSubexpression suspended while semantics change }
end;
ModeSymbol: ⟨Parse mode as radix type 1526⟩;
StructureSymbol: ⟨Parse structure as radix type 1527⟩;
othercases begin *MissingWord*(*paWrongRadTypeBeg*); *gSubexpPtr*↑.*InsertIncorrType* **end**;
endcases;
⟨Close the parentheses 1529⟩;
gSubexpPtr↑.*FinishType*;
end;

1526. ⟨ Parse mode as radix type 1526 ⟩ ≡
  begin *lSymbol* ← *CurWord.Nr*; *ReadWord*;
  if *CurWord.Kind* = *sy_Of* **then**
    if *ModeMaxArgs.fList*↑[*lSymbol*] = 0 **then** *WrongWord*(*paUnexpOf*)
    else begin *ReadWord*; *GetArguments*(*ModeMaxArgs.fList*↑[*lSymbol*]) **end**;
  end

This code is used in section 1525.

1527. ⟨ Parse structure as radix type 1527 ⟩ ≡
  begin *lSymbol* ← *CurWord.Nr*; *ReadWord*;
  if *CurWord.Kind* = *sy_Over* **then**
    if *StructModeMaxArgs.fList*↑[*lSymbol*] = 0 **then** *WrongWord*(*paUnexpOver*)
    else begin *ReadWord*; *GetArguments*(*StructModeMaxArgs.fList*↑[*lSymbol*]) **end**;
  end

This code is used in section 1525.

1528. ⟨ Parse optional left-paren 1528 ⟩ ≡
  if *CurWord.Kind* = *sy_LeftParanthesis* **then**
    begin *gSubexpPtr*↑.*ProcessLeftParenthesis*; *ReadWord*; *inc*(*lParenthCnt*);
    end

This code is used in section 1525.

1529. ⟨ Close the parentheses 1529 ⟩ ≡
  if *lParenthCnt* > 0 **then**
    begin *gSubexpPtr*↑.*ProcessRightParenthesis*; *Accept*(*sy_RightParanthesis*, *paRightParenthExp1*);
    end

This code is used in section 1525.

1530. Now we have to parse the type subexpression. We basically get the adjectives with *GetAdjectiveCluster*, then we get the radix type with *RadixTypeSubexpression*.

⟨ Parse expressions (`parser.pas`) 1502 ⟩ +≡
procedure *TypeSubexpression*;
  begin *gExpPtr*↑.*CreateSubexpression*; *gSubexpPtr*↑.*StartType*; *gSubexpPtr*↑.*StartAttributes*;
  GetAdjectiveCluster; *RadixTypeSubexpression*;
  gSubexpPtr↑.*CompleteAttributes*; *gSubexpPtr*↑.*CompleteType*;
  KillSubexpression;
  end;

1531. We should recall from Figure 8 (§1508) that this is a critical part of parsing terms.

⟨ Parse expressions (`parser.pas`) 1502 ⟩ +≡
procedure *TermSubexpression*;
  var *lParenthCnt*: *integer*;
  begin *gExpPtr*↑.*CreateSubexpression*; *OpenParenth*(*lParenthCnt*); { (§1503) }
  case *CurWord*.*Kind* **of**
  Identifier, *Numeral*, *LeftCircumfixSymbol*, *sy_It*, *sy_LeftCurlyBracket*, *sy_LeftSquareBracket*, *sy_The*,
      sy_Dolar, *StructureSymbol*:
    begin *GetClosedSubterm*; *CompleteArgument*(*lParenthCnt*); **end**;
  InfixOperatorSymbol: { skip } ;
  othercases begin *gSubexpPtr*↑.*InsertIncorrTerm*; *gSubexpPtr*↑.*FinishArgument*;
    WrongWord(*paWrongTermBeg*);
    end;
  endcases;
  AppendFunc(*lParenthCnt*); { (§1519) }
  while *lParenthCnt* > 0 **do** ⟨ Parse arguments to the right 1532 ⟩;
  gSubexpPtr↑.*FinishTerm*; *KillSubexpression*;
  end;

1532. ⟨ Parse arguments to the right 1532 ⟩ ≡
  begin *ArgumentsTail*(*MaxVisArgNbr* − 1); *dec*(*lParenthCnt*); *gSubexpPtr*↑.*ProcessRightParenthesis*;
  Accept(*sy_RightParanthesis*, *paRightParenthExp10*);
  if *CurWord*.*Kind* ≠ *InfixOperatorSymbol* **then** *MissingWord*(*paFunctExp3*);
  AppendFunc(*lParenthCnt*);
  end

This code is used in section 1531.

1533. This will parse *fArgsNbr* comma separated terms. It's used to parse the arguments "to the right" of a term, for parsing the contents of an enumerated set (e.g., $\{x, y, z, w\}$), among many other places.

We should recall that the *StartArgument* method is empty.

⟨ Parse expressions (`parser.pas`) 1502 ⟩ +≡
procedure *ArgumentsTail*(*fArgsNbr* : *integer*);
  begin while (*fArgsNbr* > 0) ∧ *Occurs*(*sy_Comma*) **do**
    begin *gSubexpPtr*↑.*StartArgument*; *TermSubexpression*; *gSubexpPtr*↑.*FinishArgument*;
    dec(*fArgsNbr*);
    end;
  end;

1534. Attributes, terms, predicates have terms as arguments. This relies upon the *FinishArguments* method (§1405).

⟨ Parse expressions (`parser.pas`) 1502 ⟩ +≡
procedure *GetArguments*(**const** *fArgsNbr*: *integer*);
  begin if *fArgsNbr* > 0 **then**
    begin *TermSubexpression*; *gSubexpPtr*↑.*FinishArgument*; *ArgumentsTail*(*fArgsNbr* − 1);
    end;
  end;

Subsection 24.1.2. Formulas

1535. Quantified variables looks like

$$\langle \mathit{Variable} \rangle \ \{ \ \texttt{","} \ \langle \mathit{Variable} \rangle \} \ [(\texttt{"be"}|\texttt{"being"}) \ \langle \mathit{Type} \rangle]$$

The parsing routine follows the grammar fairly faithfully.

⟨ Parse expressions (**parser.pas**) 1502 ⟩ +≡
procedure *QuantifiedVariables*;
  begin repeat *gSubexpPtr*↑.*StartQualifiedSegment*; *ReadWord*;
    ⟨ Parse comma-separated variables for quantified variables 1536 ⟩;
    gSubexpPtr↑.*StartQualifyingType*;
    if *Occurs*(*sy_Be*) ∨ *Occurs*(*sy_Being*) **then** *TypeSubexpression*;
    gSubexpPtr↑.*FinishQualifiedSegment*;
  until *CurWord*.*Kind* ≠ *sy_Comma*;
  end;

1536. ⟨ Parse comma-separated variables for quantified variables 1536 ⟩ ≡
  repeat *gSubexpPtr*↑.*ProcessVariable*; *Accept*(*Identifier*, *paIdentExp2*);
  until ¬*Occurs*(*sy_Comma*)

This code is used in section 1535.

1537. The existential formula looks like

$$\texttt{ex} \ \langle \mathit{Quantified\text{-}Variables} \rangle \ \texttt{st} \ \langle \mathit{Formula} \rangle$$

The parser implements it quite faithfully.

⟨ Parse expressions (**parser.pas**) 1502 ⟩ +≡
procedure *ExistentialFormula*;
  begin *gSubexpPtr*↑.*StartExistential*; *QuantifiedVariables*;
  gSubexpPtr↑.*FinishQuantified*; *Accept*(*sy_St*, *paStExp*); *FormulaSubexpression*;
  gSubexpPtr↑.*FinishExistential*;
  end;

1538. Universally quantified formulas are tricky because both

for ⟨*Quantified-Variables*⟩ holds ⟨*Formula*⟩

and

for ⟨*Quantified-Variables*⟩ st ⟨*Formula*⟩ holds ⟨*Formula*⟩

are acceptable. Furthermore, we may include multiple "for ⟨*Quantified-Variables*⟩" (possibly with "st ⟨*Formula*⟩" restrictions) before arriving at the single "holds ⟨*Formula*⟩". The trick is to parse this as

for ⟨*Quantified-Variables*⟩ [st ⟨*Formula*⟩] [holds] ⟨*Formula*⟩

so the recursive call to parse the final formula enables us to parse another quantified formula.

⟨ Parse expressions (parser.pas) 1502 ⟩ +≡
procedure *UniversalFormula*;
  begin *gSubexpPtr*↑.*StartUniversal*; *QuantifiedVariables*; *gSubexpPtr*↑.*FinishQuantified*;
  if *CurWord*.*Kind* = *sy_St* **then**
    begin *gSubexpPtr*↑.*StartRestriction*; *ReadWord*; *FormulaSubexpression*;
    gSubexpPtr↑.*FinishRestriction*;
    end;
  case *CurWord*.*Kind* **of**
  sy_Holds: **begin** *gSubexpPtr*↑.*ProcessHolds*; *ReadWord* **end**;
  sy_For, *sy_Ex*: ; { fallthrough }
  othercases begin *gSubexpPtr*↑.*InsertIncorrFormula*; *MissingWord*(*paWrongScopeBeg*) **end**;
  endcases;
  FormulaSubexpression; *gSubexpPtr*↑.*FinishUniversal*;
  end;

1539. The Parser's current token is either "=" or a predicate symbol. Then we should parse "the right-hand side" of the equation (or formula). The current token's Symbol number is passed as the argument to this procedure.

It's worth recalling the definition of *TermBegSys* (§1502) which is all the token kinds for starting a term. If the next token is a term, then *GetArguments* is invoked to parse them.

⟨ Parse expressions (parser.pas) 1502 ⟩ +≡
procedure *ConditionalTail*; *forward*;

procedure *CompleteRightSideOfThePredicativeFormula*(*aPredSymbol* : *integer*);
  begin *gSubexpPtr*↑.*ProcessRightSideOfPredicateSymbol*; *ReadWord*;
  if *CurWord*.*Kind* ∈ *TermBegSys* **then** *GetArguments*(*PredMaxArgs*.*fList*↑[*aPredSymbol*]);
  gSubexpPtr↑.*FinishRightSideOfPredicativeFormula*;
  end;

1540. Recall a "multi-predicative formula" is something of the form $a \le x \le b$. More generally, we could imagine the grammar for such a formula resembles:

$$\langle \textit{Formula} \rangle \; \{ \; \langle \textit{Multi-Predicate} \rangle \; \langle \textit{Term-List} \rangle \; \}$$

The Parser's current token is ⟨*Multi-Predicate*⟩, 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").

⟨ Parse expressions (`parser.pas`) 1502 ⟩ +≡
procedure *CompleteMultiPredicativeFormula*;
  begin *gSubexpPtr*↑.*StartMultiPredicativeFormula*;
  repeat case *CurWord*.*Kind* **of**
    sy_Equal, *PredicateSymbol*: *CompleteRightSideOfThePredicativeFormula*(*CurWord*.*Nr*);
    sy_Does, *sy_Do*: ⟨ Parse multi-predicate with "`does`" or "`do`" in copula 1541 ⟩;
    endcases;
  until ¬(*CurWord*.*Kind* ∈ [*sy_Equal*, *PredicateSymbol*, *sy_Does*, *sy_Do*]);
  gSubexpPtr↑.*FinishMultiPredicativeFormula*;
  end;

1541. ⟨ Parse multi-predicate with "`does`" or "`do`" in copula 1541 ⟩ ≡
  begin ⟨ Consume "`does not`" or "`do not`", raise error otherwise 1542 ⟩;
  if *CurWord*.*Kind* ∈ [*PredicateSymbol*, *sy_Equal*] **then**
    begin *CompleteRightSideOfThePredicativeFormula*(*CurWord*.*Nr*); *gSubexpPtr*↑.*ProcessNegative*; **end**
  else begin *gSubExpPtr*↑.*InsertIncorrFormula*; *SynErr*(*CurPos*, *paInfinitiveExp*) **end**;
  end

This code is used in section 1540.

1542. ⟨ Consume "`does not`" or "`do not`", raise error otherwise 1542 ⟩ ≡
  gSubexpPtr↑.*ProcessDoesNot*; *ReadWord*; *Accept*(*sy_Not*, *paNotExpected*)

This code is used in section 1541.

1543. The Parser is trying to parse a predicate and has just parsed a comma-separated list of terms. Now, the Parser's is either (1) looking at a predicate or equality, or (2) has matched "`does not`" or "`do not`" and is now looking at a predicate or equality. In both cases, the Parser tries to complete the formula with the *CompletePredicativeFormula* procedure.

⟨ Parse expressions (`parser.pas`) 1502 ⟩ +≡
procedure *CompletePredicativeFormula*(*aPredSymbol* : *integer*);
  begin *gSubexpPtr*↑.*ProcessPredicateSymbol*; { (§1460) }
  ReadWord;
  if *CurWord*.*Kind* ∈ *TermBegSys* **then** *GetArguments*(*PredMaxArgs*.*fList*↑[*aPredSymbol*]);
  gSubexpPtr↑.*FinishPredicativeFormula*;
  end;

1544.

⟨ Parse expressions (`parser.pas`) 1502 ⟩ +≡
procedure *CompleteAtomicFormula*(**var** *aParenthCnt* : *integer*);
  var *lPredSymbol*: *integer*;
  label *Predicate*; { not actually used }
  begin ⟨ Parse left arguments in a formula 1545 ⟩;
  case *CurWord.Kind* **of**
  sy_Equal, *PredicateSymbol*: ⟨ Parse equation or (possibly infixed) predicate 1546 ⟩;
  sy_Does, *sy_Do*: ⟨ Parse formula with "`does not`" or "`do not`" 1547 ⟩;
  sy_Is: ⟨ Parse formula with "`is not`" or "`is not`" 1548 ⟩;
  othercases begin *gSubexpPtr*↑.*ProcessAtomicFormula*; *MissingWord*(*paWrongPredSymbol*);
    gSubexpPtr↑.*InsertIncorrBasic*;
    end;
  endcases;
  end;

1545. ⟨ Parse left arguments in a formula 1545 ⟩ ≡
  repeat *AppendFunc*(*aParenthCnt*);
    if *CurWord.Kind* = *sy_Comma* **then**
      begin *ArgumentsTail*(*MaxVisArgNbr* − 1);
      if (*aParenthCnt* > 0) ∧ (*CurWord.Kind* = *sy_RightParanthesis*) **then**
        begin *dec*(*aParenthCnt*); *gSubexpPtr*↑.*ProcessRightParenthesis*; *ReadWord*;
        if *CurWord.Kind* ≠ *InfixOperatorSymbol* **then** *MissingWord*(*paFunctExp1*);
        end;
      end;
  until *CurWord.Kind* ≠ *InfixOperatorSymbol*

This code is used in section 1544.

1546. ⟨ Parse equation or (possibly infixed) predicate 1546 ⟩ ≡
  begin *CompletePredicativeFormula*(*CurWord.Nr*);
  if *CurWord.Kind* ∈ [*sy_Equal*, *PredicateSymbol*, *sy_Does*, *sy_Do*] **then** *CompleteMultiPredicativeFormula*
  end

This code is used in section 1544.

1547. ⟨ Parse formula with "`does not`" or "`do not`" 1547 ⟩ ≡
  begin *gSubexpPtr*↑.*ProcessDoesNot*; *ReadWord*; *Accept*(*sy_Not*, *paNotExpected*);
  if *CurWord.Kind* ∈ [*PredicateSymbol*, *sy_Equal*] **then**
    begin *CompletePredicativeFormula*(*CurWord.Nr*); *gSubexpPtr*↑.*ProcessNegative*;
    if *CurWord.Kind* ∈ [*sy_Equal*, *PredicateSymbol*, *sy_Does*, *sy_Do*] **then**
      CompleteMultiPredicativeFormula
    end
  else begin *gSubExpPtr*↑.*InsertIncorrFormula*; *SynErr*(*CurPos*, *paInfinitiveExp*) **end**;
  end

This code is used in section 1544.

1548. ⟨ Parse formula with "`is not`" or "`is not`" 1548 ⟩ ≡

begin *gSubexpPtr*↑.*ProcessAtomicFormula*; *ReadWord*;
if (*CurWord.Kind* = *sy_Not*) ∧ (*AheadWord.Kind* ∈ *TermBegSys* + [*ModeSymbol*, *StructureSymbol*, *sy_Set*, *AttributeSymbol*, *sy_Non*]) ∨ (*CurWord.Kind* ∈ *TermBegSys* + [*ModeSymbol*, *StructureSymbol*, *sy_Set*, *AttributeSymbol*, *sy_Non*]) **then**
  begin *gSubexpPtr*↑.*StartType*; *gSubexpPtr*↑.*StartAttributes*;
  if *CurWord.Kind* = *sy_Not* **then**
    begin *gSubexpPtr*↑.*ProcessNegation*; *ReadWord*; **end**;
  GetAdjectiveCluster;
  case *CurWord.Kind* **of**
  sy_LeftParanthesis, *ModeSymbol*, *StructureSymbol*, *sy_Set*: **begin** *RadixTypeSubexpression*;
    gSubexpPtr↑.*CompleteAttributes*; *gSubexpPtr*↑.*CompleteType*;
    gSubexpPtr↑.*FinishQualifyingFormula*;
    end;
  othercases begin *gSubexpPtr*↑.*CompleteAttributes*; *gSubexpPtr*↑.*FinishAttributiveFormula*; **end**;
  endcases;
  end
else begin *gSubExpPtr*↑.*InsertIncorrFormula*; *WrongWord*(*paTypeOrAttrExp*); **end**;
end

This code is used in section 1544.

1549. There is a comment in Polish, a single word ("Kolejnosc") which translates into English as "Order".

define *starts_with_term_token* ≡ *Numeral*, *LeftCircumfixSymbol*, *sy_It*, *sy_LeftCurlyBracket*, *sy_LeftSquareBracket*, *sy_The*, *sy_Dolar*, *StructureSymbol*

⟨ Parse expressions (`parser.pas`) 1502 ⟩ +≡
procedure *ViableFormula*;
  var *lParenthCnt*: *integer*;
  label *NotPrivate*;
  begin *gExpPtr*↑.*CreateSubexpression*; *OpenParenth*(*lParenthCnt*);
  case *CurWord.Kind* **of**
  sy_For: *UniversalFormula*;
  sy_Ex: *ExistentialFormula*; { !!!!!!!!!!!!!!! Order }
  sy_Contradiction: **begin** *gSubexpPtr*↑.*ProcessContradiction*; *ReadWord*; **end**;
  sy_Thesis: **begin** *gSubexpPtr*↑.*ProcessThesis*; *ReadWord*; **end**;
  sy_Not: **begin** *gSubexpPtr*↑.*ProcessNot*; *ReadWord*; *ViableFormula*; *KillSubexpression*;
    gSubexpPtr↑.*ProcessNegative*;
    end;
  Identifier: **if** *AheadWord.Kind* = *sy_LeftSquareBracket* **then** ⟨ Parse private formula 1550 ⟩
    else goto *NotPrivate*;
  starts_with_term_token:
    NotPrivate: **begin** *gSubexpPtr*↑.*StartAtomicFormula*; { ??? TermSubexpression }
      GetClosedSubterm; *CompleteArgument*(*lParenthCnt*); *CompleteAtomicFormula*(*lParenthCnt*);
      end;
  InfixOperatorSymbol, *PredicateSymbol*, *sy_Does*, *sy_Do*, *sy_Equal*: **begin** *gSubexpPtr*↑.*StartAtomicFormula*;
    CompleteAtomicFormula(*lParenthCnt*);
    end;
  othercases begin *gSubexpPtr*↑.*InsertIncorrFormula*; *WrongWord*(*paWrongFormulaBeg*) **end**;
  endcases; ⟨ Close parentheses for formula 1551 ⟩;
  end;

1550. ⟨ Parse private formula 1550 ⟩ ≡
begin $gSubexpPtr\uparrow.StartPrivateFormula$; $ReadWord$; $ReadWord$;
if $CurWord.Kind \neq sy\_RightSquareBracket$ **then** $GetArguments(MaxVisArgNbr)$;
$Accept(sy\_RightSquareBracket, paRightSquareExp2)$; $gSubexpPtr\uparrow.FinishPrivateFormula$;
end

This code is used in section 1549.

1551. ⟨ Close parentheses for formula 1551 ⟩ ≡
while $lParenthCnt > 0$ **do**
begin $ConditionalTail$; $gSubexpPtr\uparrow.ProcessRightParenthesis$;
$Accept(sy\_RightParanthesis, paRightParenthExp4)$; $dec(lParenthCnt)$; $CloseParenth(lParenthCnt)$;
end

This code is used in section 1549.

1552. Precedence for logical connectives. We will now "hardcode" the precedence for logical 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`".

⟨ Parse expressions (`parser.pas`) 1502 ⟩ +≡
procedure $ConjunctiveTail$;
begin while $(CurWord.Kind = sy\_Ampersand) \wedge (AheadWord.Kind \neq sy\_Ellipsis)$ **do**
begin $gSubexpPtr\uparrow.ProcessBinaryConnective$; $ReadWord$; $ViableFormula$; $KillSubexpression$;
$gSubexpPtr\uparrow.FinishBinaryFormula$;
end;
end;

1553. Mizar parses flexary conjunctions ("$\Phi[0]$ `& ... &` $\Phi[n]$") as weaker than "ordinary conjunction". For example "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.

⟨ Parse expressions (`parser.pas`) 1502 ⟩ +≡
procedure $FlexConjunctiveTail$;
begin $ConjunctiveTail$;
if $CurWord.Kind = sy\_Ampersand$ **then**
begin $Assert(AheadWord.Kind = sy\_Ellipsis)$; $ReadWord$; $ReadWord$; $Accept(sy\_Ampersand, 402)$;
$gSubexpPtr\uparrow.ProcessFlexConjunction$; $ViableFormula$; $ConjunctiveTail$; $KillSubexpression$;
$gSubexpPtr\uparrow.FinishFlexConjunction$;
end;
end;

1554. Disjunction binds weaker than flexary conjunction (which binds weaker than ordinary conjunction).

As for ordinary conjunction, Mizar parses multiple disjunctions as left associative. So "`A or B or C`" parses as "`(A or B) or C`".

⟨ Parse expressions (`parser.pas`) 1502 ⟩ +≡
procedure *DisjunctiveTail*;
  begin *FlexConjunctiveTail*;
  while (*CurWord.Kind* = *sy_Or*) ∧ (*AheadWord.Kind* ≠ *sy_Ellipsis*) **do**
    begin *gSubexpPtr*↑.*ProcessBinaryConnective*; *ReadWord*; *ViableFormula*; *FlexConjunctiveTail*;
    KillSubexpression; *gSubexpPtr*↑.*FinishBinaryFormula*;
    end;
  end;

1555. Parsing a disjunction will have the Parser's current token be "`or`" only if the next token is an ellipsis ("`...`"), which is precisely the signal for a flexary disjunction. When the current token is not an "`or`", then the Parser does nothing (its work is done).

When the user forgets an "`or`" after ellipsis (e.g., writing "`A or ... C`"), a 401 error will be raised.

⟨ Parse expressions (`parser.pas`) 1502 ⟩ +≡
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;

1556. Mizar parses "`implies`" and "`iff`" with lower precedence than "`or`", matching common Mathematical practice. Working Mathematicians read "`A or B implies C`" as "`(A or B) implies C`". We impose this precedence with the *FlexDisjunctiveTail* parsing the remaining disjunctions before checking for "`iff`" or "`implies`".

Mizar accepts one "topmost" implication connective. So "`A implies B implies C`" would be illegal (a 336 error would be raised). You would have to insert parentheses to make this parseable by Mizar (i.e., "`A implies (B implies C)`"). This makes sense for implication, but there is a compelling argument that "`A iff B iff C`" could be parsed as "`(A iff B) & (B iff C)`" — that latter formula *could* be parsed properly by Mizar.

⟨ Parse expressions (`parser.pas`) 1502 ⟩ +≡
procedure *ConditionalTail*;
  begin *FlexDisjunctiveTail*;
  case *CurWord.Kind* **of**
  sy_Implies, *sy_Iff* : **begin** *gSubexpPtr*↑.*ProcessBinaryConnective*; *ReadWord*; *ViableFormula*;
    FlexDisjunctiveTail; *KillSubexpression*; *gSubexpPtr*↑.*FinishBinaryFormula*;
    case *CurWord.Kind* **of**
    sy_Implies, *sy_Iff* : *WrongWord*(*paUnexpConnective*);
    endcases;
    end;
  endcases;
  end;

1557. Formula subexpressions. When the Parser needs a formula as a subexpression for a larger expression — like when it parses a Fraenkel term (an expression), the Parser will need to parse

$$\{\langle \textit{Term}\rangle\ \langle \textit{Qualifying-Segment}\rangle\ \texttt{:}\ \langle \textit{Formula-Subexpression}\rangle\}$$

This will also serve as the "workhorse" for parsing a formula expression.

⟨ Parse expressions (`parser.pas`) 1502 ⟩ +≡

procedure *FormulaSubexpression*;
  begin *ViableFormula*; *ConditionalTail*; *KillSubexpression*;
  end;

Section 24.2. COMMUNICATION WITH ITEMS

1558. When the Parser constructs the AST for a term, the workflow is as follows:

(1) Allocate a new *extExpression* object, and update *gExprPtr* to point at it.
(2) Using the *gExprPtr* to allocate a new *extSubexp* object, and update the *gSubexpPtr* to point at it.
(3) The Parser will invoke methods for the *gSubexpPtr*'s reference to build the AST. The result will be stored in a state variable (like *gLastTerm* or *gLastType*).
(4) There will be residual objects allocated, stored in the fields of *gSubexpPtr* and *gExpPtr*. We need to clean those up, freeing them, by invoking *KillExpression* and *KillSubexpression*.

So each of these methods have the following template: allocate a new expression object, update the *gExpPtr* to point to it, parse something, then free the *gExpPtr* using the *KillExpression* procedure.

⟨ Communicate with items (`parser.pas`) 1558 ⟩ ≡

{ *Communication with items* }

procedure *TermExpression*;
  begin *gItemPtr*↑.*CreateExpression*(*exTerm*); *TermSubexpression*; *KillExpression*;
  end;

procedure *TypeExpression*;
  begin *gItemPtr*↑.*CreateExpression*(*exType*); *TypeSubexpression*; *KillExpression*;
  end;

procedure *FormulaExpression*;
  begin *gItemPtr*↑.*CreateExpression*(*exFormula*); *FormulaSubexpression*; *KillExpression*;
  end;

This code is used in section 1493.

Section 24.3. MISCELLANEOUS

1559. When the Parser is looking at a label, the *gItemPtr* will construct the label. The Parser still needs to move past the "⟨*identifier*⟩`:`" two tokens.

⟨ Process miscellany (`parser.pas`) 1559 ⟩ ≡
  { *Miscellaneous* }
procedure *ProcessLab*;
  begin *gItemPtr*↑.*ProcessLabel*; { (§1365) }
  if (*CurWord*.*Kind* = *Identifier*) ∧ (*AheadWord*.*Kind* = *sy_Colon*) **then**
    begin *ReadWord*; *ReadWord* **end**;
  end;

See also sections 1560, 1561, 1562, 1563, 1564, 1565, 1567, 1568, 1569, and 1570.
This code is used in section 1493.

1560. Telling the *gItemPtr* state variable we are about to parse a sentence just invokes the *StartSentence* (§1280) method, then the Parser parses the formula, and the *gItemPtr* "finishes" the sentence (which is an empty method).

⟨ Process miscellany (`parser.pas`) 1559 ⟩ +≡
procedure *ProcessSentence*;
  begin *gItemPtr*↑.*StartSentence*; *FormulaExpression*; *gItemPtr*↑.*FinishSentence*;
  end;

1561. When the Parser expected a sentence but something unexpected happened, specifically an unexpected statement has cross the Parser's path. When that statement has encountered an unjustified "`per cases`". We just create a new formula expression, and specifically an "incorrect formula".

⟨ Process miscellany (`parser.pas`) 1559 ⟩ +≡
procedure *InCorrSentence*;
  begin *gItemPtr*↑.*StartSentence*; *gItemPtr*↑.*CreateExpression*(*exFormula*);
  gExpPtr↑.*CreateSubexpression*; *gSubexpPtr*↑.*InsertIncorrFormula*; *KillSubexpression*; *KillExpression*;
  gItemPtr↑.*FinishSentence*;
  end;

1562. The Parser attempts to recover (or at least, report) an unexpected item when expecting a statement. Specifically, a "`per cases`" appears when it should not.

⟨ Process miscellany (`parser.pas`) 1559 ⟩ +≡
procedure *InCorrStatement*;
  begin *gItemPtr*↑.*ProcessLabel*; *gItemPtr*↑.*StartRegularStatement*; *InCorrSentence*;
  end;

1563. The Parser is looking at either

$$\texttt{let}\ \langle\mathit{Variables}\rangle\ \texttt{being}\ \langle\mathit{Type}\rangle\ \texttt{such that}\ \langle\mathit{Hypotheses}\rangle$$

or

$$\texttt{assume that}\ \langle\mathit{Hypotheses}\rangle$$

Specifically, the Parser has arrived at the "⟨*Hypotheses*⟩" bit and needs to parse it. The ⟨*Hypotheses*⟩ generically looks like

$$\langle\mathit{Hypotheses}\rangle = [\langle\mathit{label}\rangle]\ \langle\mathit{Formula}\rangle\ \{\ \texttt{and}\ \langle\mathit{Hypotheses}\rangle\ \}$$

That is to say, a bunch of (possibly labeled) formulas joined together by "`and`" keywords.

⟨ Process miscellany (`parser.pas`) 1559 ⟩ +≡
procedure *ProcessHypotheses*;
  begin repeat *ProcessLab*; *ProcessSentence*; *gItemPtr*↑.*FinishHypothesis*;
  until ¬*Occurs*(*sy_And*)
  end;

1564. An assumption is either collective (using hypotheses) or singular (a single, possibly labeled, formula).

⟨ Process miscellany (`parser.pas`) 1559 ⟩ +≡
procedure *Assumption*;
  begin if *CurWord.Kind* = *sy_That* **then**
    begin *gItemPtr*↑.*StartCollectiveAssumption*; *ReadWord*; *ProcessHypotheses*
    end
  else begin *ProcessLab*; *ProcessSentence*; *gItemPtr*↑.*FinishHypothesis*;
    end;
  gItemPtr↑.*FinishAssumption*;
  end;

1565. Existential elimination in Mizar looks like

$$\texttt{consider}\ \langle\mathit{Fixed\text{-}variables}\rangle\ \texttt{such that}\ \langle\mathit{Formula}\rangle$$

The ⟨*Fixed-variables*⟩ is just a comma-separated list of segments.

⟨ Process miscellany (`parser.pas`) 1559 ⟩ +≡
procedure *FixedVariables*;
  begin *gItemPtr*↑.*StartFixedVariables*;
  repeat ⟨ Parse segment of fixed variables 1566 ⟩;
  until ¬*Occurs*(*sy_Comma*);
  gItemPtr↑.*FinishFixedVariables*;
  end;

1566. And a "fixed" segment is just a comma-separated list of variables. This is either implicitly qualified (i.e., they are all reserved variables) or explicitly qualified (i.e., there is a "`being`" or "`be`", followed by a type). A 300 error will be raised if the comma-separated list of variables encounters something other than an identifier.

⟨ Parse segment of fixed variables 1566 ⟩ ≡
  gItemPtr↑.*StartFixedSegment*;
  repeat *gItemPtr*↑.*ProcessFixedVariable*; *Accept*(*Identifier*, *paIdentExp4*);
  until ¬*Occurs*(*sy_Comma*);
  gItemPtr↑.*ProcessBeing*; { parse the type qualification }
  if *Occurs*(*sy_Be*) ∨ *Occurs*(*sy_Being*) **then** *TypeExpression*;
  gItemPtr↑.*FinishFixedSegment*

This code is used in section 1565.

1567. The Parser is trying to parse a "`consider`" statement or a "`given`" statement. The Parser will try to parse

⟨*Fixed-Variables*⟩ `such that` ⟨*Formula*⟩ { `and` ⟨*Formula*⟩ }

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.

⟨ Process miscellany (`parser.pas`) 1559 ⟩ +≡
procedure *ProcessChoice*;
  begin *FixedVariables*; *Accept*(*sy_Such*, *paSuchExp*); *Accept*(*sy_That*, *paThatExp2*);
  repeat *gItemPtr*↑.*StartCondition*; *ProcessLab*; *ProcessSentence*; *gItemPtr*↑.*FinishCondition*;
  until ¬*Occurs*(*sy_And*);
  gItemPtr↑.*FinishChoice*;
  end;

1568. The Parser is looking at the "`let`" token. There are two possible statements

`let` ⟨*Fixed-variables*⟩`;`

or possibly with assumptions

`let` ⟨*Fixed-variables*⟩ `such that` ⟨*Hypotheses*⟩`;`

If the user forgot "`that`" but included a "`such`" after the fixed-variables, a 350 error is raised.

⟨ Process miscellany (`parser.pas`) 1559 ⟩ +≡
procedure *Generalization*;
  begin *ReadWord*; *FixedVariables*;
  if *Occurs*(*sy_Such*) **then**
    begin *gItemPtr*↑.*StartAssumption*; *Accept*(*sy_That*, *paThatExp1*); *ProcessHypotheses*;
    gItemPtr↑.*FinishAssumption*;
    end;
  end;

1569. The Parser is looking at the "`given`" token currently. This is the same as "`assume ex` $\vec{x}$ `st` $\Phi[\vec{x}]$`;` `then consider` $\vec{x}$ `such that` $\Phi[\vec{x}]$`;`".

⟨ Process miscellany (`parser.pas`) 1559 ⟩ +≡
procedure *ExistentialAssumption*;
  begin *gBlockPtr*↑.*CreateItem*(*itExistentialAssumption*); *ReadWord*; *ProcessChoice*;
  end;

1570. The Parser is looking at either "`canceled;`" or "`canceled` ⟨*number*⟩`;`".

⟨ Process miscellany (`parser.pas`) 1559 ⟩ +≡
procedure *Canceled*;
  begin *gBlockPtr*↑.*CreateItem*(*itCanceled*); *ReadWord*;
  if *CurWord*.*Kind* = *Numeral* **then** *ReadWord*;
  gItemPtr↑.*FinishTheorem*;
  end;

Section 24.4. SIMPLE JUSTIFICATIONS

1571. The Parser is looking at "`by`" and now needs to parse the list of references. If the user tries to use something other than a label's identifier as a reference, then a 308 error will be raised.

⟨ Parse simple justifications (`parser.pas`) 1571 ⟩ ≡
  { *Simple Justifications* }
procedure *GetReferences*;
  begin *gItemPtr*↑.*StartReferences*;
  repeat *ReadWord*; ⟨ Parse single reference 1572 ⟩;
  until *CurWord*.*Kind* ≠ *sy_Comma*;
  gItemPtr↑.*FinishReferences*;
  end;

See also sections 1574 and 1576.

This code is used in section 1493.

1572. ⟨ Parse single reference 1572 ⟩ ≡
  case *CurWord*.*Kind* **of**
  MMLIdentifier: ⟨ Parse library references 1573 ⟩;
  Identifier: **begin** *gItemPtr*↑.*ProcessPrivateReference*; *ReadWord* **end**;
  othercases *WrongWord*(*paWrongReferenceBeg*);
  endcases

This code is used in section 1571.

1573. Mizar supports multiple references from the same article to "piggieback" off the same article "anchor". For example, "`GROUP_1:13,def 3,17`" refers to theorems 13 and 17 and definition 3 from the MML Article `GROUP_1`.

If the user forgot to include the theorem or definition number — so they just wrote "⟨*Article*⟩" instead of "⟨*Article*⟩`:`⟨*Number*⟩" or "⟨*Article*⟩`:def` ⟨*Number*⟩" — then Mizar flags this with a 384 error.

  define *no_longer_referencing_article* ≡ (*CurWord*.*Kind* ≠ *sy_Comma*) ∨
      (*AheadWord*.*Kind* = *Identifier*) ∨ (*AheadWord*.*Kind* = *MMLIdentifier*)

⟨ Parse library references 1573 ⟩ ≡
  begin *gItemPtr*↑.*StartLibraryReferences*; *ReadWord*;
  if *CurWord*.*Kind* = *sy_Colon* **then**
    repeat *ReadWord*; *gItemPtr*↑.*ProcessDef*;
      if *CurWord*.*Kind* = *ReferenceSort* **then**
        begin if *CurWord*.*Nr* ≠ *ord*(*syDef*) **then** *ErrImm*(*paDefExp*);
        ReadWord;
        end;
      gItemPtr↑.*ProcessTheoremNumber*; *Accept*(*Numeral*, *paNumExp*);
    until *no_longer_referencing_article*
  else *MissingWord*(*paColonExp4*);
  gItemPtr↑.*FinishTheLibraryReferences*;
  end

This code is used in section 1572.

1574. The Parser is currently looking at "`from`", which means a reference to a scheme identifier will be given next (possibly followed with a comma-separated list of references in parentheses).

If the user tries to give something else (instead of an identifier of a scheme), then a 308 error will be raised. Also, if the user forgot the closing parentheses around the references for the scheme (e.g., "`from MyScheme(A1,A2`"), then 370 error will be raised.

⟨ Parse simple justifications (`parser.pas`) 1571 ⟩ +≡
procedure *GetSchemeReference*;
  begin *gItemPtr*↑.*StartSchemeReference*; *ReadWord*;
  case *CurWord.Kind* **of**
  MMLIdentifier: ⟨ Parse reference to scheme from MML 1575 ⟩;
  Identifier: **begin** *gItemPtr*↑.*ProcessSchemeReference*; *ReadWord* **end**;
  othercases *WrongWord*(*paWrongReferenceBeg*);
  endcases;
  if *CurWord.Kind* = *sy_LeftParanthesis* **then**
    begin *GetReferences*; *Accept*(*sy_RightParanthesis*, *paRightParenthExp7*)
    end;
  gItemPtr↑.*FinishSchemeReference*;
  end;

1575. Mizar expects scheme references to the MML to be of the form "`from` ⟨*Article*⟩`:sch` ⟨*Number*⟩". 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.

⟨ Parse reference to scheme from MML 1575 ⟩ ≡
  begin *gItemPtr*↑.*StartSchemeLibraryReference*; *ReadWord*;
  if *CurWord.Kind* = *sy_Colon* **then**
    begin *ReadWord*; *gItemPtr*↑.*ProcessSch*;
    if *CurWord.Kind* = *ReferenceSort* **then**
      begin if *CurWord.Nr* ≠ *ord*(*sySch*) **then** *ErrImm*(*paSchExp*);
      ReadWord;
      end
    else *ErrImm*(*paSchExp*);
    gItemPtr↑.*ProcessSchemeNumber*; *Accept*(*Numeral*, *paNumExp*);
    end
  else *MissingWord*(*paColonExp4*);
  gItemPtr↑.*FinishSchLibraryReferences*;
  end

This code is used in section 1574.

1576. The Parser expects a simple justification — i.e., either a "`by`" followed by some references, or "`from`" followed by a scheme reference. For some "obvious" inferences, no justification may be needed.

⟨ Parse simple justifications (`parser.pas`) 1571 ⟩ +≡
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;

Section 24.5. STATEMENTS AND REASONINGS

1577. Pragmas have been enabled which tells Mizar to skip the proof. The Parser simply stores a counter (initialized to 1), and increments it every time a "`proof`" token has been encountered, but decrements it every time an "`end`" token has been encountered. When the counter has reached zero, the proof has ended, and the Parser can stop skipping things.

There are, of course, other blocks which use "`end`" to terminate it. For example, definitions. But if the Parser should encounter such tokens, then things have gone so horribly awry, the Parser should just quit here and now.

⟨Parse statements and reasoning (`parser.pas`) 1577⟩ ≡

```
   { Statements & Reasonings }
procedure Reasoning; forward;

procedure IgnoreProof;
  var lCounter: integer; ReasPos: Position;
  begin gBlockPtr↑.StartAtSignProof; ReasPos ← CurPos; ReadTokenProc; lCounter ← 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↑.FinishAtSignProof;
  end;
```

See also sections 1578, 1579, 1580, 1585, 1586, 1591, 1598, and 1599.

This code is used in section 1493.

1578. Parsing either a "`by`" justification (or a "`from`" justification) or a nested "`proof`" block. If the Parser is looking at neither situation, the *SimpleJustification* procedure will raise errors.

```
  define parse_proof ≡
          if ProofPragma then Reasoning
          else IgnoreProof
```

⟨Parse statements and reasoning (`parser.pas`) 1577⟩ +≡

```
procedure Justification;
  begin gItemPtr↑.StartJustification;
  case CurWord.Kind of
  sy_Proof: parse_proof;
  othercases SimpleJustification;
  endcases; gItemPtr↑.FinishJustification;
  end;
```

1579. For private predicates ("`defpred`") and private functors ("`deffunc`"), there will be a list of comma-separated types for the arguments of the private definition.

define *parse_comma_separated_types* ≡
  repeat *TypeExpression*; *gItemPtr*↑.*FinishLocusType*
  until ¬*Occurs*(*sy_Comma*)

⟨ Parse statements and reasoning (`parser.pas`) 1577 ⟩ +≡
procedure *ReadTypeList*;
  begin case *CurWord*.*Kind* **of**
  sy_RightSquareBracket, *sy_RightParanthesis*: ;
  othercases *parse_comma_separated_types*;
  endcases;
  end;

1580. A **"Private Item"** is a statement ("item") which introduces a new constant local ("private") to the block or article.

define *other_regular_statements* ≡ *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*, *StructureSymbol*, *sy_The*, *sy_LeftCurlyBracket*, *sy_Proof*

⟨ Parse statements and reasoning (`parser.pas`) 1577 ⟩ +≡
procedure *RegularStatement*; *forward*; { (§1599) }
procedure *PrivateItem*;
  begin *gBlockPtr*↑.*ProcessLink*;
  if *CurWord*.*Kind* = *sy_Then* **then** *ReadWord*;
  case *CurWord*.*Kind* **of**
  sy_Deffunc: ⟨ Parse a "`deffunc`" 1581 ⟩;
  sy_Defpred: ⟨ Parse a "`defpred`" 1582 ⟩;
  sy_Set: ⟨ Parse a "`set`" constant definition 1583 ⟩;
  sy_Reconsider: ⟨ Parse a "`reconsider`" statement 1584 ⟩;
  sy_Consider: **begin** *gBlockPtr*↑.*CreateItem*(*itChoice*); *ReadWord*; *ProcessChoice*; *SimpleJustification*;
    end;
  other_regular_statements: **begin** *gBlockPtr*↑.*CreateItem*(*itRegularStatement*); *RegularStatement*; **end**;
  othercases begin *gBlockPtr*↑.*CreateItem*(*itIncorrItem*); *WrongWord*(*paWrongItemBeg*); **end**;
  endcases;
  end;

1581. ⟨ Parse a "`deffunc`" 1581 ⟩ ≡
  begin *gBlockPtr*↑.*CreateItem*(*itPrivFuncDefinition*); *ReadWord*; *gItemPtr*↑.*StartPrivateDefiniendum*;
  Accept(*Identifier*, *paIdentExp6*); *Accept*(*sy_LeftParanthesis*, *paLeftParenthExp*); *ReadTypeList*;
  Accept(*sy_RightParanthesis*, *paRightParenthExp8*); *gItemPtr*↑.*StartPrivateDefiniens*;
  Accept(*sy_Equal*, *paEqualityExp1*); *TermExpression*; *gItemPtr*↑.*FinishPrivateFuncDefinienition*;
  end

This code is used in section 1580.

1582. ⟨ Parse a "`defpred`" 1582 ⟩ ≡
  begin *gBlockPtr*↑.*CreateItem*(*itPrivPredDefinition*); *ReadWord*; *gItemPtr*↑.*StartPrivateDefiniendum*;
  Accept(*Identifier*, *paIdentExp7*); *Accept*(*sy_LeftSquareBracket*, *paLeftSquareExp*); *ReadTypeList*;
  Accept(*sy_RightSquareBracket*, *paRightSquareExp4*); *gItemPtr*↑.*StartPrivateDefiniens*;
  Accept(*sy_Means*, *paMeansExp*); *FormulaExpression*; *gItemPtr*↑.*FinishPrivatePredDefinienition*;
  end

This code is used in section 1580.

1583. ⟨ Parse a "`set`" constant definition 1583 ⟩ ≡
begin *gBlockPtr*↑.*CreateItem*(*itConstantDefinition*); *ReadWord*;
repeat *gItemPtr*↑.*StartPrivateConstant*; *Accept*(*Identifier*, *paIdentExp8*);
Accept(*sy_Equal*, *paEqualityExp2*); *TermExpression*; *gItemPtr*↑.*FinishPrivateConstant*;
until ¬*Occurs*(*sy_Comma*);
end

This code is used in section 1580.

1584. ⟨ Parse a "`reconsider`" statement 1584 ⟩ ≡
begin *gBlockPtr*↑.*CreateItem*(*itReconsider*); *ReadWord*;
repeat *gItemPtr*↑.*ProcessReconsideredVariable*; *Accept*(*Identifier*, *paIdentExp9*);
case *CurWord*.*Kind* **of**
sy_Equal: **begin** *ReadWord*; *TermExpression*; *gItemPtr*↑.*FinishReconsideredTerm*;
end;
else *gItemPtr*↑.*FinishDefaultTerm*;
end;
until ¬*Occurs*(*sy_Comma*);
gItemPtr↑.*StartNewType*; *Accept*(*sy_As*, *paAsExp*); *TypeExpression*; *gItemPtr*↑.*FinishReconsidering*;
SimpleJustification;
end

This code is used in section 1580.

1585. The *SetParserPragma* toggles the state variables for skipping proofs, and storing the pragma in the AST is handled by the *gBlockPtr*'s method call.

⟨ Parse statements and reasoning (`parser.pas`) 1577 ⟩ +≡
procedure *ProcessPragmas*;
begin while *CurWord*.*Kind* = *Pragma* **do**
begin *SetParserPragma*(*CurWord*.*Spelling*); { (§1195) }
gBlockPtr↑.*ProcessPragma*; { (§1215) }
ReadTokenProc;
end;
end;

1586. 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`) 1577 ⟩ +≡
procedure *LinearReasoning*;
begin while *CurWord*.*Kind* ≠ *sy_End* **do**
begin *StillCorrect* ← *true*; *ProcessPragmas*; ⟨ Parse statement of linear reasoning 1587 ⟩;
Semicolon;
end;
end;

1587. Most statements are delegated to their own dedicated function.

⟨ Parse statement of linear reasoning 1587 ⟩ ≡
  case *CurWord.Kind* **of**
  sy_Let: **begin** *gBlockPtr*↑.*CreateItem*(*itGeneralization*); *Generalization*; **end**;
  sy_Given: *ExistentialAssumption*;
  sy_Assume: **begin** *gBlockPtr*↑.*CreateItem*(*itAssumption*); *ReadWord*; *Assumption*; **end**;
  sy_Take: ⟨ Parse "`take`" statement for linear reasoning 1588 ⟩;
  sy_Hereby: **begin** *gBlockPtr*↑.*CreateItem*(*itConclusion*); *Reasoning*; **end**;
  ⟨ Parse "`thus`" and "`hence`" for linear reasoning 1589 ⟩;
  sy_Per: *exit*;
  sy_Case, *sy_Suppose*: *exit*;
  sy_Reserve, *sy_Scheme*, *sy_Theorem*, *sy_Definition*, *sy_Begin*, *sy_Notation*, *sy_Registration*, *EOT*: *exit*;
  sy_Then: ⟨ Parse "`then`" for linear reasoning 1590 ⟩;
  othercases *PrivateItem*;
  endcases

This code is used in section 1587.

1588. Take statements. We recall the syntax for a "`take`" statement:

$$\texttt{take}\ (\langle\mathit{Term}\rangle \mid \langle\mathit{Variable}\rangle = \langle\mathit{Term}\rangle)\ \{\texttt{","}\ (\langle\mathit{Term}\rangle \mid \langle\mathit{Variable}\rangle = \langle\mathit{Term}\rangle)\}$$

That is, a comma-separated list of either (1) terms, or (2) a variable equal to a term.

⟨ Parse "`take`" statement for linear reasoning 1588 ⟩ ≡
  begin *gBlockPtr*↑.*CreateItem*(*itExemplification*); *ReadWord*;
  repeat if (*CurWord.Kind* = *Identifier*) ∧ (*AheadWord.Kind* = *sy_Equal*) **then**
      begin *gItemPtr*↑.*ProcessExemplifyingVariable*; *ReadWord*; *ReadWord*; *TermExpression*;
      gItemPtr↑.*FinishExemplifyingVariable*;
      end
    else begin *gItemPtr*↑.*StartExemplifyingTerm*; *TermExpression*; *gItemPtr*↑.*FinishExemplifyingTerm*;
      end;
  until ¬*Occurs*(*sy_Comma*);
  end

This code is used in section 1587.

1589. 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 *RegularStatement* for parsing the formula. But the *gBlockPtr* state variable "primes the pump" by creating a "conclusion" statement.

⟨ Parse "`thus`" and "`hence`" for linear reasoning 1589 ⟩ ≡
sy_Hence: **begin** *gBlockPtr*↑.*ProcessLink*; *ReadWord*; *gBlockPtr*↑.*CreateItem*(*itConclusion*);
  RegularStatement;
  end;
sy_Thus: **begin** *ReadWord*; *gBlockPtr*↑.*ProcessLink*;
  if *CurWord.Kind* = *sy_Then* **then** *ReadWord*;
  gBlockPtr↑.*CreateItem*(*itConclusion*); *RegularStatement*;
  end

This code is used in section 1587.

1590. Parsing 'then' linked statements.

⟨ Parse "`then`" for linear reasoning 1590 ⟩ ≡
  begin if *AheadWord.Kind* = *sy_Per* **then**
    begin *gBlockPtr*↑.*ProcessLink*; *ReadWord*; *exit*; **end**
  else *PrivateItem*;
  end

This code is used in section 1587.

1591. Non-block Reasoning. The Parser has just encountered a "`per cases`" statement. Now it must parse "`suppose`" items.

⟨ Parse statements and reasoning (`parser.pas`) 1577 ⟩ +≡
procedure *NonBlockReasoning*;
  var *CasePos*: *Position*; *lCaseKind*: *TokenKind*; ⟨ Process "`case`" (local procedure) 1592 ⟩;
  begin case *CurWord.Kind* **of**
  sy_Per, *sy_Case*, *sy_Suppose*: **begin** *gBlockPtr*↑.*CreateItem*(*itPerCases*);
    ⟨ Consume "`per cases`", raise an error if they're missing 1593 ⟩;
    if $(CurWord.Kind \neq sy\_Case) \land (CurWord.Kind \neq sy\_Suppose)$ **then**
      ⟨ Try to synchronize after failing to find initial '`case`' or '`suppose`' 1594 ⟩;
    repeat ⟨ Parse "`suppose`" or "`case`" block 1595 ⟩;
    until (*Curword.Kind* = *sy_End*);
    end;
  endcases;
  end;

1592. Each "`case`" or "`suppose`" block consists of zero or more linear reasoning items, followed possibly by an optional "non-block reasoning" proof (i.e., another nested "`per cases`" proof by cases).

⟨ Process "`case`" (local procedure) 1592 ⟩ ≡
procedure *ProcessCase*;
  begin *Assumption*; *Semicolon*; *LinearReasoning*;
  if *CurWord.Kind* = *sy_Per* **then** *NonBlockReasoning*;
  KillBlock; *AcceptEnd*(*CasePos*); *Semicolon*;
  end

This code is used in section 1591.

1593. 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.

⟨ Consume "`per cases`", raise an error if they're missing 1593 ⟩ ≡
  Accept(*sy_Per*, *paPerExp*); *Accept*(*sy_Cases*, *paCasesExp*); *SimpleJustification*; *Semicolon*;
  lCaseKind ← *CurWord.Kind*

This code is used in section 1591.

1594. The Parser is expecting "`suppose`" or "`case`" after the "`per cases`" statement. But if the Parser fails to find either of these tokens, it *should* enter panic mode.

Like a person falling off a cliff reaches out for something to grab, the Parser in panic mode seeks something to "grab on to" so the Parser can "soldier on". The technical term for this situation is that the Parser is trying to "synchronize" (usually people just talk about "synchronization").

Mizar raises a 232 error.

⟨ Try to synchronize after failing to find initial '`case`' or '`suppose`' 1594 ⟩ ≡
begin *MissingWord*(*paSupposeOrCaseExp*); *lCaseKind* ← *sy_Suppose*;
gBlockPtr↑.*CreateItem*(*itCaseBlock*); *gBlockPtr*↑.*CreateBlock*(*blSuppose*);
gBlockPtr↑.*CreateItem*(*itSupposeHead*); *StillCorrect* ← *true*; *CasePos* ← *CurPos*; *ProcessCase*;
end

This code is used in section 1591.

1595. ⟨ Parse "`suppose`" or "`case`" block 1595 ⟩ ≡
while (*CurWord*.*Kind* = *sy_Case*) ∨ (*CurWord*.*Kind* = *sy_Suppose*) **do**
⟨ Parse contents of "`suppose`" block 1596 ⟩;
case *Curword*.*Kind* **of**
sy_Reserve, *sy_Scheme*, *sy_Theorem*, *sy_Definition*, *sy_Begin*, *sy_Notation*, *sy_Registration*, *EOT*: *exit*;
sy_End: ;
othercases ⟨ Synchronize after missing '`suppose`' or '`case`' token 1597 ⟩;
endcases

This code is used in section 1591.

1596. Parsing the contents of a "`suppose`" or "`case`" block requires creating a new block (for the, you know, block) and creating a new item for the "`suppose` ⟨*Formula*⟩" or "`case` ⟨*Formula*⟩" statement.

If the user tries to "mix and match" the different kind of suppositions (i.e., "`case`" and "`suppose`"), then a 58 error should be raised.

define *create_supposition_block* ≡
if *lCaseKind* = *sy_Case* **then** *gBlockPtr*↑.*CreateBlock*(*blCase*)
else *gBlockPtr*↑.*CreateBlock*(*blSuppose*)
define *create_supposition_head* ≡
if *lCaseKind* = *sy_Case* **then** *gBlockPtr*↑.*CreateItem*(*itCaseHead*)
else *gBlockPtr*↑.*CreateItem*(*itSupposeHead*)

⟨ Parse contents of "`suppose`" block 1596 ⟩ ≡
begin *gBlockPtr*↑.*CreateItem*(*itCaseBlock*); *create_supposition_block*; *CasePos* ← *CurPos*;
StillCorrect ← *true*; *create_supposition_head*;
if *CurWord*.*Kind* ≠ *lCaseKind* **then** *ErrImm*(58);
ReadWord; *ProcessCase*;
end

This code is used in section 1595.

1597. ⟨ Synchronize after missing '`suppose`' or '`case`' token 1597 ⟩ ≡
begin *MissingWord*(*paSupposeOrCaseExp*); *gBlockPtr*↑.*CreateItem*(*itCaseBlock*);
create_supposition_block; *create_supposition_head*; *StillCorrect* ← *true*; *CasePos* ← *CurPos*;
ProcessCase;
end

This code is used in section 1595.

1598. 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).

⟨ Parse statements and reasoning (`parser.pas`) 1577 ⟩ +≡
procedure *Reasoning*;
  var *ReasPos*: *Position*;
  begin *ReasPos* ← *CurPos*;
  case *CurWord.Kind* **of**
  sy_Proof: **begin** *gBlockPtr*↑.*CreateBlock*(*blProof*); *ReadTokenProc*; **end**;
  sy_Hereby: **begin** *gBlockPtr*↑.*CreateBlock*(*blHereby*); *ReadTokenProc*; **end**;
  sy_Now: **begin** *gBlockPtr*↑.*CreateBlock*(*blDiffuse*); *ReadTokenProc*; **end**;
  othercases begin *gBlockPtr*↑.*CreateBlock*(*blProof*); *WrongWord*(*paProofExp*); **end**;
  endcases;
  LinearReasoning; *NonBlockReasoning*; *KillBlock*; *AcceptEnd*(*ReasPos*);
  end;

1599. Regular statements. A regular statement is one of the following:
(1) "`now`" followed by reasoning;
(2) A sentence (i.e., possibly labeled formula) followed by a "`proof`" block;
(3) Iterative equalities.

⟨ Parse statements and reasoning (`parser.pas`) 1577 ⟩ +≡
procedure *RegularStatement*;
  begin *ProcessLab*; *gItemPtr*↑.*StartRegularStatement*;
  case *CurWord.Kind* **of**
  sy_Now: *Reasoning*;
  othercases begin *ProcessSentence*;
    case *CurWord.Kind* **of**
    sy_Proof: ⟨ Parse "`proof`" block 1600 ⟩;
    othercases begin *gItemPtr*↑.*StartJustification*; *SimpleJustification*; *gItemPtr*↑.*FinishJustification*;
      gItemPtr↑.*FinishCompactStatement*;
      while *CurWord.Kind* = *sy_DotEquals* **do** ⟨ Parse iterative equations 1601 ⟩;
      end;
    endcases;
    end;
  endcases;
  end;

1600. ⟨ Parse "`proof`" block 1600 ⟩ ≡
  begin *gItemPtr*↑.*StartJustification*;
  if *ProofPragma* **then** *Reasoning*
  else *IgnoreProof*;
  gItemPtr↑.*FinishJustification*;
  end

This code is used in section 1599.

1601. ⟨ Parse iterative equations 1601 ⟩ ≡
  begin *gItemPtr*↑.*StartIterativeStep*; *ReadWord*; *TermExpression*; *gItemPtr*↑.*ProcessIterativeStep*;
  gItemPtr↑.*StartJustification*; *SimpleJustification*; *gItemPtr*↑.*FinishJustification*;
  gItemPtr↑.*FinishIterativeStep*;
  end

This code is used in section 1599.

Section 24.6. PATTERNS

1602. Visible arguments (compared to "hidden arguments") appear to the left or right of a functor or predicate (or to the left of an attribute, or to the right of a mode or structure). The *gVisibleNbr* state variable is initialized to zero when the Parser starts parsing visible arguments, and the Parser increments it for each visible argument in the pattern.

If a non-identifier appears in a pattern, Mizar raises a 300 error. So you cannot be clever and try to trick Mizar into thinking "`0 + x`" is a pattern.

⟨ Parse patterns (`parser.pas`) 1602 ⟩ ≡
  { *Patterns* }
var *gVisibleNbr*: *integer*;
procedure *GetVisible*;
  begin *gItemPtr*↑.*ProcessVisible*;   { (§1325) }
  inc(*gVisibleNbr*); *Accept*(*Identifier*, *paIdentExp3*);
  end;

See also sections 1603, 1604, 1607, 1608, 1609, 1612, 1614, and 1615.
This code is used in section 1493.

1603. We will need to Parse a comma-separated list of identifiers when determining a pattern.

⟨ Parse patterns (`parser.pas`) 1602 ⟩ +≡
procedure *ReadVisible*;
  begin *gItemPtr*↑.*StartVisible*; *gVisibleNbr* ← 0;
  repeat *GetVisible*;
  until ¬*Occurs*(*sy_Comma*);
  gItemPtr↑.*FinishVisible*;
  end;

1604. There are two cases to consider when determining the pattern for a mode: either the Parser is looking at "`set`" as a type, or—the more interesting case—the Parser is looking at an identifier which appears in a vocabulary file as a mode symbol.

⟨ Parse patterns (`parser.pas`) 1602 ⟩ +≡
procedure *GetModePattern*;
  var *lModesymbol*: *integer*;
  begin *gItemPtr*↑.*StartModePattern*;   { (§1313) }
  case *CurWord*.*Kind* **of**
  sy_Set: ⟨ Parse pattern for "`set`" as a mode 1605 ⟩;
  ModeSymbol: ⟨ Parse pattern for a mode symbols 1606 ⟩
  othercases *WrongWord*(*paWrongModePatternBeg*);
  endcases;
  gItemPtr↑.*FinishModePattern*;   { (§1314) }
  end;

1605. ⟨ Parse pattern for "`set`" as a mode 1605 ⟩ ≡
  begin if *AheadWord*.*Kind* = *sy_Of* **then** *WrongWord*(*paWrongModePatternSet*)
  else *ReadWord*;
  end

This code is used in section 1604.

1606. The "⟨*Kind*⟩*MaxArgs*" entry is initialized to \$*FF* before *ReadVisible* is invoked, which is PASCAL for $^\#$FF = 255. So if the *ModeMaxArgs* entry for the mode symbol is (1) less than the number of arguments parsed, or (2) uninitialized; then we should update its entry with the *gVisibleNbr* state variable's current value.

define *get_index_compare_to_default*(#) ≡ [#] = \$*FF*
define *entry_is_unitialized*(#) ≡ #.*fList*↑*get_index_compare_to_default*

⟨ Parse pattern for a mode symbols 1606 ⟩ ≡
begin *lModeSymbol* ← *CurWord.Nr*; *gVisibleNbr* ← 0; *ReadWord*; *gItemPtr*↑.*ProcessModePattern*;
if *Occurs*(*sy_Of*) **then** *ReadVisible*;
if (*ModeMaxArgs.fList*↑[*lModeSymbol*] < *gVisibleNbr*) ∨
(*entry_is_uninitialized*(*ModeMaxArgs*)(*lModeSymbol*)) **then**
ModeMaxArgs.fList↑[*lModeSymbol*] ← *gVisibleNbr*;
end

This code is used in section 1604.

1607. Parsing the visible arguments for a functor relies on this helper function.

⟨ Parse patterns (parser.pas) 1602 ⟩ +≡
procedure *ReadParams*;
begin if *Occurs*(*sy_LeftParanthesis*) **then**
begin *ReadVisible*; *Accept*(*sy_RightParanthesis*, *paRightParenthExp5*) **end**
else if *CurWord.Kind* = *Identifier* **then**
begin *gItemPtr*↑.*StartVisible*; *GetVisible*; *gItemPtr*↑.*FinishVisible*; **end**;
end;

1608. Attribute patterns allows for arguments *only on the right* of the attribute symbol, i.e., something like

$$\texttt{attr}\ \underbrace{\langle\textit{Identifier}\rangle\ \texttt{is}\ \langle\textit{Arguments}\rangle\ \langle\textit{Attribute-Name}\rangle}_{\text{pattern}}\ \texttt{means}\ldots$$

⟨ Parse patterns (parser.pas) 1602 ⟩ +≡
procedure *GetAttrPattern*;
begin *gItemPtr*↑.*StartAttributePattern*; *gVisibleNbr* ← 0; *GetVisible*;
gItemPtr↑.*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↑.*FinishAttributePattern*; *Accept*(*AttributeSymbol*, *paAttrExp2*);
end;

1609. Functor patterns generically look like:

$$\texttt{func}\ \underbrace{\langle \textit{Arguments} \rangle\ \langle \textit{Identifier} \rangle\ \langle \textit{Arguments} \rangle}_{\text{pattern}}\ \texttt{->}\ldots$$

or

$$\texttt{func}\ \underbrace{\langle \textit{Left-Bracket} \rangle\ \langle \textit{Arguments} \rangle\ \langle \textit{Right-Bracket} \rangle}_{\text{pattern}}\ \texttt{->}\ldots$$

⟨ Parse patterns (`parser.pas`) 1602 ⟩ +≡
procedure *GetFuncPattern*;
  begin *gItemPtr*↑.*StartFunctorPattern*;
  case *CurWord*.*Kind* **of**
  Identifier, *InfixOperatorSymbol*, *sy_LeftParanthesis*: ⟨ Parse infix functor pattern 1610 ⟩;
  LeftCircumfixSymbol, *sy_LeftSquareBracket*, *sy_LeftCurlyBracket*: ⟨ Parse bracket functor pattern 1611 ⟩;
  othercases begin *WrongWord*(*paWrongFunctorPatternBeg*); *gItemPtr*↑.*FinishFunctorPattern*; **end**;
  endcases;
  end;

1610. ⟨ Parse infix functor pattern 1610 ⟩ ≡
  begin *ReadParams*; *gItemPtr*↑.*ProcessFunctorSymbol*; { (§1322) }
  Accept(*InfixOperatorSymbol*, *paFunctExp2*); *ReadParams*; *gItemPtr*↑.*FinishFunctorPattern*;
  end

This code is used in section 1609.

1611. ⟨ Parse bracket functor pattern 1611 ⟩ ≡
  begin *ReadWord*; *ReadVisible*; *gItemPtr*↑.*FinishFunctorPattern*;
  case *Curword*.*Kind* **of**
  sy_RightSquareBracket, *sy_RightCurlyBracket*, *sy_RightParanthesis*: *ReadWord*;
  othercases *Accept*(*RightCircumfixSymbol*, *paRightBraExp2*);
  endcases;
  end

This code is used in section 1609.

1612. Predicate patterns resemble infix functor patterns.

⟨ Parse patterns (`parser.pas`) 1602 ⟩ +≡
procedure *GetPredPattern*;
  var *lPredSymbol*: *integer*;
  begin *gItemPtr*↑.*StartPredicatePattern*;
  if *CurWord*.*Kind* = *Identifier* **then** *ReadVisible*;
  gItemPtr↑.*ProcessPredicateSymbol*;
  case *CurWord*.*Kind* **of**
  sy_Equal, *PredicateSymbol*: ⟨ Parse predicate pattern 1613 ⟩;
  othercases *WrongWord*(*paWrongPredPattern*);
  endcases; *gItemPtr*↑.*FinishPredicatePattern*;
  end;

1613. ⟨ Parse predicate pattern 1613 ⟩ ≡

begin *lPredSymbol* ← *CurWord.Nr*;
if *CurWord.Kind* = *sy_Equal* **then** *lPredSymbol* ← *EqualitySym*;
gVisibleNbr ← 0; *ReadWord*;
if *CurWord.Kind* = *Identifier* **then** *ReadVisible*;
if (*PredMaxArgs.fList*↑[*lPredSymbol*] < *gVisibleNbr*) ∨ (*entry_is_uninitialized*(*PredMaxArgs*)(*lPredSymbol*))
  then *PredMaxArgs.fList*↑[*lPredSymbol*] ← *gVisibleNbr*;
end

This code is used in section 1612.

1614. The "specification" (appearing in a non-expandable mode and functor definitions) refers to the "`->` ⟨*Type*⟩" portion which gives the type for the functor or mode.

⟨ Parse patterns (`parser.pas`) 1602 ⟩ +≡

procedure *Specification*;
begin *gItemPtr*↑.*StartSpecification*; *Accept*(*sy_Arrow*, *paArrowExp1*); *TypeExpression*;
gItemPtr↑.*FinishSpecification*;
end;

1615. Parsing a structure pattern is a bit misleading. Unlike the previous procedures, this will actually parse the entirety of a structure definition:

`struct` ⟨*Identifier*⟩ `(` ⟨*Types*⟩ `)` `(#` ⟨*Fields*⟩ `#)`

⟨ Parse patterns (`parser.pas`) 1602 ⟩ +≡

procedure *GetStructPatterns*;
var *lStructureSymbol*: *integer*;
begin *gBlockPtr*↑.*CreateItem*(*itDefStruct*); *ReadWord*;
⟨ Parse ancestors of structure, if there are any 1616 ⟩;
⟨ Parse "`over`" and any structure arguments, if any 1617 ⟩;
gItemPtr↑.*StartFields*;
⟨ Update max arguments for structure symbol, if needed 1618 ⟩;
⟨ Parse the fields of the structure definition 1619 ⟩;
end;

1616. ⟨ Parse ancestors of structure, if there are any 1616 ⟩ ≡

if *CurWord.Kind* = *sy_LeftParanthesis* **then**
  begin repeat *gItemPtr*↑.*StartPrefix*; *ReadWord*; *TypeExpression*; *gItemPtr*↑.*FinishPrefix*;
  until *CurWord.Kind* ≠ *sy_Comma*;
  Accept(*sy_RightParanthesis*, *paRightParenthExp6*);
  end

This code is used in section 1615.

1617. ⟨ Parse "`over`" and any structure arguments, if any 1617 ⟩ ≡

gItemPtr↑.*ProcessStructureSymbol*; *lStructureSymbol* ← `$`*FF*;
if *CurWord.Kind* = *StructureSymbol* **then** *lStructureSymbol* ← *CurWord.Nr*;
Accept(*StructureSymbol*, *paStructExp1*);
if *Occurs*(*sy_Over*) **then** *ReadVisible*

This code is used in section 1615.

1618. ⟨ Update max arguments for structure symbol, if needed 1618 ⟩ ≡
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 1615.

1619. ⟨ Parse the fields of the structure definition 1619 ⟩ ≡
$Accept(sy\_StructLeftBracket, paLeftDoubleExp3)$;
repeat ⟨ Parse field for the structure definition 1620 ⟩;
until $\neg Occurs(sy\_Comma)$;
$gItemPtr\uparrow.FinishFields$; $Accept(sy\_StructRightBracket, paRightDoubleExp2)$

This code is used in section 1615.

1620. ⟨ Parse field for the structure definition 1620 ⟩ ≡
$gItemPtr\uparrow.StartAggrPattSegment$;
repeat $gItemPtr\uparrow.ProcessField$; $Accept(SelectorSymbol, paSelectExp1)$;
until $\neg Occurs(sy\_Comma)$;
$Specification$; $gItemPtr\uparrow.FinishAggrPattSegment$

This code is used in section 1619.

Section 24.7. DEFINITIONS

1621. Non-expandable modes, i.e., modes of the form

`mode` ⟨*Identifier*⟩ `of` ⟨*Arguments*⟩ `->` ⟨*Type*⟩ `means` ⟨*Formula*⟩

⟨ Parse definitions (`parser.pas`) 1621 ⟩ ≡

```
    { Definitions }
procedure ConstructionType;
  begin gItemPtr↑.StartConstructionType;   { (§1364) }
  if CurWord.Kind = sy_Arrow then
    begin ReadWord; TypeExpression end;
  gItemPtr↑.FinishConstructionType;   { (§1306) }
  end;
```

See also sections 1622, 1623, 1630, 1631, 1632, 1633, 1634, 1638, 1641, 1643, 1647, 1648, 1649, 1650, 1651, 1652, and 1653.
This code is used in section 1493.

1622. Parsing correctness conditions amounts to looping through every "⟨*Correctness*⟩ ⟨*Justification*⟩`;`" statement, with a fallback "`correctness` ⟨*Justification*⟩`;`" correctness condition.

There is a comment, "o jaki tu item chodzi? definitional-item?", which Google translates from Polish as, "What item are we talking about here? Definitional-item?" I have swapped this into the code snippet.

⟨ Parse definitions (`parser.pas`) 1621 ⟩ +≡

```
procedure Correctness;
  begin while CurWord.Kind = sy_CorrectnessCondition do
    begin StillCorrect ← true; gBlockPtr↑.CreateItem(itCorrCond); ReadWord; Justification;
    Semicolon;
    end;
  gItemPtr↑.ProcessCorrectness;   { (§1363) What item are we talking about here? Definitional-item? }
  if CurWord.Kind = sy_Correctness then   { "correctness" catchall }
    begin StillCorrect ← true; gBlockPtr↑.CreateItem(itCorrectness); ReadWord; Justification;
    Semicolon;
    end;
  end;
```

1623.

⟨ Parse definitions (`parser.pas`) 1621 ⟩ +≡
procedure *Definition*;
  var *lDefKind*: *TokenKind*; *lDefiniensExpected*: *boolean*;
  begin *lDefKind* ← *CurWord.Kind*; *lDefiniensExpected* ← *true*;
  case *CurWord.Kind* **of**
  sy_Mode: ⟨ Parse mode definition 1624 ⟩;
  sy_Attr: **begin** *gBlockPtr*↑.*CreateItem*(*itDefAttr*); *ReadWord*; *GetAttrPattern*; **end**;
  sy_Struct: **begin** *GetStructPatterns*; *lDefiniensExpected* ← *false*; **end**;
  sy_Func: **begin** *gBlockPtr*↑.*CreateItem*(*itDefFunc*); *ReadWord*; *GetFuncPattern*; *ConstructionType*;
    end;
  sy_Pred: **begin** *gBlockPtr*↑.*CreateItem*(*itDefPred*); *ReadWord*; *gItemPtr*↑.*StartDefPredicate*;
    GetPredPattern;
    end;
  endcases;
  if *lDefiniensExpected* **then** ⟨ Parse definiens 1625 ⟩;
  Semicolon; *Correctness*;
  while (*CurWord.Kind* = *sy_Property*) **do**
    begin *gBlockPtr*↑.*CreateItem*(*itProperty*); *StillCorrect* ← *true*; *ReadWord*; *Justification*; *Semicolon*;
    end;
  gBlockPtr↑.*FinishDefinition*;
  end;

1624. ⟨ Parse mode definition 1624 ⟩ ≡
  begin *gBlockPtr*↑.*CreateItem*(*itDefMode*); *ReadWord*; *GetModePattern*;
  case *CurWord.Kind* **of**
  sy_Is: **begin** *gItemPtr*↑.*StartExpansion*; *ReadWord*; *TypeExpression*; *lDefiniensExpected* ← *false*;
    end;
  othercases *ConstructionType*;
  endcases;
  end

This code is used in section 1623.

1625. ⟨ Parse definiens 1625 ⟩ ≡
  case *CurWord.Kind* **of**
  sy_Means: ⟨ Parse "`means`" definiens 1626 ⟩;
  sy_Equals: ⟨ Parse "`equals`" definiens 1628 ⟩;
  endcases

This code is used in section 1623.

1626. ⟨ Parse "**means**" definiens 1626 ⟩ ≡

```
begin gItemPtr↑.ProcessMeans; ReadWord;
if Occurs(sy_Colon) then
  begin gItemPtr↑.ProcessDefiniensLabel; Accept(Identifier, paIdentExp10);
  Accept(sy_Colon, paColonExp2);
  end
else gItemPtr↑.ProcessDefiniensLabel;
gItemPtr↑.StartDefiniens; FormulaExpression;
if CurWord.Kind = sy_If then ⟨ Parse "means" definition-by-cases 1627 ⟩
else gItemPtr↑.FinishOtherwise;
gItemPtr↑.FinishDefiniens;
end
```

This code is used in section 1625.

1627. ⟨ Parse "means" definition-by-cases 1627 ⟩ ≡

```
begin gItemPtr↑.StartGuard; ReadWord; FormulaExpression; gItemPtr↑.FinishGuard;
while Occurs(sy_Comma) do
  begin FormulaExpression; gItemPtr↑.StartGuard; Accept(sy_If, paIfExp); FormulaExpression;
  gItemPtr↑.FinishGuard;
  end;
if CurWord.Kind = sy_Otherwise then
  begin gItemPtr↑.StartOtherwise; ReadWord; FormulaExpression; gItemPtr↑.FinishOtherwise; end;
end
```

This code is used in section 1626.

1628. ⟨ Parse "**equals**" definiens 1628 ⟩ ≡

```
if lDefKind ≠ 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↑.ProcessDefiniensLabel;
  gItemPtr↑.StartEquals; TermExpression;
  if CurWord.Kind = sy_If then ⟨ Parse "equals" definition-by-cases 1629 ⟩
  else gItemPtr↑.FinishOtherwise;
  gItemPtr↑.FinishDefiniens;
  end
```

This code is used in section 1625.

1629. ⟨ Parse "equals" definition-by-cases 1629 ⟩ ≡

```
begin gItemPtr↑.StartGuard; ReadWord; FormulaExpression; gItemPtr↑.FinishGuard;
while Occurs(sy_Comma) do
  begin TermExpression; gItemPtr↑.StartGuard; Accept(sy_If, paIfExp); FormulaExpression;
  gItemPtr↑.FinishGuard;
  end;
if CurWord.Kind = sy_Otherwise then
  begin gItemPtr↑.StartOtherwise; ReadWord; TermExpression; gItemPtr↑.FinishOtherwise;
  end;
end
```

This code is used in section 1628.

1630. 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 ≡ (CurWord.Kind = Identifier) ∧ (AheadWord.Kind = sy_Is)
define is_infix_pattern ≡ (CurWord.Kind ∈ [LeftCircumfixSymbol, sy_LeftCurlyBracket,
        sy_LeftSquareBracket, sy_LeftParanthesis, InfixOperatorSymbol]) ∨ ((CurWord.Kind =
        Identifier) ∧ (AheadWord.Kind = InfixOperatorSymbol))
define is_predicate_pattern ≡ (CurWord.Kind = PredicateSymbol) ∨ (CurWord.Kind = sy_Equal) ∨
        ((CurWord.Kind = Identifier) ∧ (AheadWord.Kind ∈ [sy_Comma, PredicateSymbol, sy_Equal]))
define is_selector_pattern ≡ (CurWord.Kind = sy_The) ∧ (AheadWord.Kind = SelectorSymbol)
define is_forgetful_functor_pattern ≡ (CurWord.Kind = sy_The) ∧ (AheadWord.Kind = StructureSymbol)
```

⟨ Parse definitions (`parser.pas`) 1621 ⟩ +≡

```
function CurrPatternKind: TokenKind;
  begin if CurWord.Kind = ModeSymbol then CurrPatternKind ← ModeSymbol
  else if CurWord.Kind = StructureSymbol then CurrPatternKind ← StructureSymbol
  else if is_attr_pattern then CurrPatternKind ← AttributeSymbol
  else if is_infix_pattern then CurrPatternKind ← InfixOperatorSymbol
  else if is_predicate_pattern then CurrPatternKind ← PredicateSymbol
  else if is_selector_pattern then CurrPatternKind ← SelectorSymbol
  else if is_forgetful_functor_pattern then CurrPatternKind ← ForgetfulFunctor
  else CurrPatternKind ← sy_Error;
  end;
```

1631. The Parser is looking at the "`synonym`" token when this procedure is invoked.

⟨ Parse definitions (`parser.pas`) 1621 ⟩ +≡

```
procedure Synonym;
  begin ReadWord;
  case CurrPatternKind of
  ModeSymbol: begin { Mode synonym }
    gBlockPtr↑.CreateItem(itModeNotation); GetModePattern; gItemPtr↑.ProcessModeSynonym;
    Accept(sy_For, paForExp); GetModePattern;
    end;
  AttributeSymbol: begin { Attribute synonym }
    gBlockPtr↑.CreateItem(itAttrSynonym); GetAttrPattern; gItemPtr↑.ProcessAttrSynonym;
    Accept(sy_For, paForExp); GetAttrPattern;
    end;
  InfixOperatorSymbol: begin { Functor synonym }
    gBlockPtr↑.CreateItem(itFuncNotation); GetFuncPattern; gItemPtr↑.ProcessFuncSynonym;
    Accept(sy_For, paForExp); GetFuncPattern;
    end;
  PredicateSymbol: begin { Predicate synonym }
    gBlockPtr↑.CreateItem(itPredSynonym); gItemPtr↑.StartDefPredicate; GetPredPattern;
    gItemPtr↑.ProcessPredSynonym; Accept(sy_For, paForExp); GetPredPattern;
    end
  othercases begin gBlockPtr↑.CreateItem(itIncorrItem); ErrImm(paWrongPattBeg1); end;
  endcases;
  end;
```

1632. Antonyms only make sense for attributes and predicates. A 314 error is raised for any other kind of antonym.

⟨ Parse definitions (`parser.pas`) 1621 ⟩ +≡
procedure *Antonym*;
  begin *ReadWord*;
  case *CurrPatternKind* **of**
  Attributesymbol: **begin** { Attribute antonym }
    gBlockPtr↑.*CreateItem*(*itAttrAntonym*); *GetAttrPattern*; *gItemPtr*↑.*ProcessAttrAntonym*;
    Accept(*sy_For*, *paForExp*); *GetAttrPattern*;
    end;
  PredicateSymbol: **begin** { Predicate antonym }
    gBlockPtr↑.*CreateItem*(*itPredAntonym*); *gItemPtr*↑.*StartDefPredicate*; *GetPredPattern*;
    gItemPtr↑.*ProcessPredAntonym*; *Accept*(*sy_For*, *paForExp*); *GetPredPattern*;
    end
  othercases begin *gBlockPtr*↑.*CreateItem*(*itIncorrItem*); *ErrImm*(*paWrongPattBeg2*); **end**;
  endcases;
  end;

1633.

⟨ Parse definitions (`parser.pas`) 1621 ⟩ +≡
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: **begin** *gBlockPtr*↑.*CreateItem*(*itIncorrItem*); *ErrImm*(*paWrongItemBeg*); *ReadWord*;
    if *CurWord*.*Kind* = *sy_Cases* **then**
      begin *ReadWord*; *InCorrStatement*; *SimpleJustification*; **end**;
    end;
  othercases begin *ErrImm*(*paUnexpItemBeg*); *StillCorrect* ← *true*; *PrivateItem*; **end**;
  endcases;
  end;

1634. The Parser is currently looking at the "`definition`" token, so it will construct a definition block AST.

⟨ Parse definitions (`parser.pas`) 1621 ⟩ +≡
procedure *DefinitionalBlock*;
  var *DefPos*: *Position*;
  begin *gBlockPtr*↑.*CreateItem*(*itDefinition*); *gBlockPtr*↑.*CreateBlock*(*blDefinition*); *DefPos* ← *CurPos*;
  ReadWord;
  while *CurWord*.*Kind* ≠ *sy_End* **do** ⟨ Parse item in definition block 1635 ⟩;
  KillBlock; *AcceptEnd*(*DefPos*);
  end;

1635. ⟨ Parse item in definition block 1635 ⟩ ≡

```
begin StillCorrect ← true; gBlockPtr↑.ProcessRedefine;
if Occurs(sy_Redefine) then ⟨ Check we are redefining a mode, attribute, functor, or predicate 1636 ⟩;
case CurWord.Kind of
sy_Mode, sy_Attr, sy_Struct, sy_Func, sy_Pred: Definition;
sy_Begin, EOT, sy_Reserve, sy_Scheme, sy_Theorem, sy_Definition, sy_Registration, sy_Notation: break;
Pragma: ProcessPragmas;
othercases begin ⟨ Parse loci, assumptions, unexpected items in a definition block 1637 ⟩;
  Semicolon;
  end;
endcases;
end
```

This code is used in section 1634.

1636. ⟨ Check we are redefining a mode, attribute, functor, or predicate 1636 ⟩ ≡

```
if ¬(CurWord.Kind ∈ [sy_Mode, sy_Attr, sy_Func, sy_Pred]) then Error(PrevPos, paUnexpRedef)
```

This code is used in section 1635.

1637. ⟨ Parse loci, assumptions, unexpected items in a definition block 1637 ⟩ ≡

```
case CurWord.Kind of
sy_Let: begin gBlockPtr↑.CreateItem(itLociDeclaration); Generalization; end;
sy_Given: ExistentialAssumption;
sy_Assume: begin gBlockPtr↑.CreateItem(itAssumption); ReadWord; Assumption; end;
sy_Canceled: Canceled;
sy_Case, sy_Suppose, sy_Per, sy_Hereby: UnexpectedItem;
othercases PrivateItem;
endcases
```

This code is used in section 1635.

1638. 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.

⟨ Parse definitions (`parser.pas`) 1621 ⟩ +≡

```
procedure NotationBlock;
  var DefPos: Position;
  begin gBlockPtr↑.CreateItem(itDefinition); gBlockPtr↑.CreateBlock(blNotation); DefPos ← CurPos;
  ReadWord;
  while CurWord.Kind ≠ sy_End do ⟨ Parse item for notation block 1639 ⟩;
  KillBlock; AcceptEnd(DefPos);
  end;
```

1639. ⟨ Parse item for notation block 1639 ⟩ ≡

```
begin StillCorrect ← true;
case CurWord.Kind of
sy_Begin, EOT, sy_Reserve, sy_Scheme, sy_Theorem, sy_Definition, sy_Registration, sy_Notation: break;
Pragma: ProcessPragmas;
othercases ⟨ Parse semicolon-separated items in a notation block 1640 ⟩;
endcases;
end
```

This code is used in section 1638.

1640. ⟨ Parse semicolon-separated items in a notation block 1640 ⟩ ≡

begin case *CurWord.Kind* **of**
sy_Synonym: *Synonym*;
sy_Antonym: *Antonym*;
sy_Let: **begin** *gBlockPtr*↑.*CreateItem*(*itLociDeclaration*); *ReadWord*; *FixedVariables*; **end**;
othercases *UnexpectedItem*;
endcases;
Semicolon;
end

This code is used in section 1639.

1641.

define *ahead_is_type* ≡ (*AheadWord.Kind* ∈ [*sy_Set*, *ModeSymbol*, *StructureSymbol*])
define *is_attr_token* ≡ (*CurWord.Kind* ∈ [*AttributeSymbol*, *sy_Non*]) ∨
  (*CurWord.Kind* ∈ (*TermBegSys* − [*sy_LeftParanthesis*, *StructureSymbol*])) ∨
  ((*CurWord.Kind* = *sy_LeftParanthesis*) ∧ ¬(*ahead_is_type*)) ∨
  (*CurWord.Kind* = *StructureSymbol*) ∧ (*AheadWord.Kind* = *sy_StructLeftBracket*)

⟨ Parse definitions (`parser.pas`) 1621 ⟩ +≡

procedure *ATTSubexpression*(**var** *aExpKind* : *ExpKind*);
  var *lAttrExp*: *boolean*;
  begin *aExpKind* ← *exNull*; *gSubexpPtr*↑.*StartAttributes*;
  while *is_attr_token* **do**
    begin *gSubexpPtr*↑.*ProcessNon*; *lAttrExp* ← *CurWord.Kind* = *sy_Non*;
    if *CurWord.Kind* = *sy_Non* **then** *ReadWord*;
    ⟨ Parse arguments for attribute expression 1642 ⟩;
    if *CurWord.Kind* = *AttributeSymbol* **then**
      begin *aExpKind* ← *exAdjectiveCluster*; *gSubexpPtr*↑.*ProcessAttribute*; *ReadWord*; **end**
    else begin if *lAttrExp* ∨ (*aExpKind* = *exAdjectiveCluster*) **then**
        { *aExpKind* = *exAdjectiveCluster* is never true }
        begin *gSubexpPtr*↑.*ProcessAttribute*; *SynErr*(*CurPos*, *paAttrExp3*); **end**;
      break;
      end;
    end;
  gSubexpPtr↑.*CompleteAttributes*;
  end;

1642. ⟨ Parse arguments for attribute expression 1642 ⟩ ≡

if (*CurWord.Kind* ∈ (*TermBegSys* − [*StructureSymbol*])) ∨
    (*CurWord.Kind* = *StructureSymbol*) ∧ (*AheadWord.Kind* = *sy_StructLeftBracket*) **then**
  begin if *aExpKind* = *exNull* **then** *aExpKind* ← *exTerm*;
  gSubexpPtr↑.*StartAttributeArguments*; *ProcessArguments*; *gSubexpPtr*↑.*FinishAttributeArguments*;
  end

This code is used in section 1641.

1643. Registration clusters.

⟨ Parse definitions (`parser.pas`) 1621 ⟩ +≡
procedure *RegisterCluster*;
  var *lExpKind*: *ExpKind*;
  begin *gBlockPtr*↑.*CreateItem*(*itCluster*); *ReadWord*;
  if (*CurWord*.*Kind* = *Identifier*) ∧ (*AheadWord*.*Kind* = *sy_Arrow*) **then** *ErrImm*(*paFunctExp4*);
  gItemPtr↑.*StartAttributes*; { (§1278) }
  gItemPtr↑.*CreateExpression*(*exAdjectiveCluster*); { (§1351) }
  gExpPtr↑.*CreateSubexpression*; *ATTSubexpression*(*lExpKind*);
  case *lExpKind* **of**
  exTerm: *gSubexpPtr*↑.*CompleteClusterTerm*;
  exNull, *exAdjectiveCluster*: *gSubexpPtr*↑.*CompleteAdjectiveCluster*;
  endcases;
  KillSubexpression; *KillExpression*;
  case *lExpKind* **of**
  exTerm: ⟨ Parse functor registration cluster 1644 ⟩;
  exNull, *exAdjectiveCluster*: **case** *CurWord*.*Kind* **of**
    sy_Arrow: ⟨ Parse conditional registration cluster 1645 ⟩;
    sy_For: ⟨ Parse existential registration cluster 1646 ⟩;
    othercases begin *SynErr*(*CurPos*, *paForOrArrowExpected*); *gItemPtr*↑.*FinishConsequent*;
      gItemPtr↑.*CreateExpression*(*exType*); *gExpPtr*↑.*CreateSubexpression*; *gSubexpPtr*↑.*StartType*;
      gSubexpPtr↑.*InsertIncorrType*; *gSubexpPtr*↑.*CompleteType*; *gSubexpPtr*↑.*CompleteClusterType*;
      KillSubexpression; *KillExpression*; *gItemPtr*↑.*FinishClusterType*;
      end;
    endcases;
  endcases; *Semicolon*; *Correctness*;
  end;

1644. ⟨ Parse functor registration cluster 1644 ⟩ ≡
  begin *gItemPtr*↑.*FinishClusterTerm*; *Accept*(*sy_Arrow*, *paArrowExp2*);
  gItemPtr↑.*CreateExpression*(*exAdjectiveCluster*); *gExpPtr*↑.*CreateSubexpression*;
  gSubexpPtr↑.*StartAttributes*; *ATTSubexpression*(*lExpKind*);
  if *lExpKind* ≠ *exAdjectiveCluster* **then**
    begin *ErrImm*(*paAdjClusterExp*) **end**;
  gSubexpPtr↑.*CompleteAdjectiveCluster*; *KillSubexpression*; *KillExpression*;
  gItemPtr↑.*FinishConsequent*;
  if *CurWord*.*Kind* = *sy_For* **then**
    begin *ReadWord*; *gItemPtr*↑.*CreateExpression*(*exType*); *gExpPtr*↑.*CreateSubexpression*;
    gSubexpPtr↑.*StartType*; *gSubexpPtr*↑.*StartAttributes*; *GetAdjectiveCluster*; *RadixTypeSubexpression*;
    gSubexpPtr↑.*CompleteAttributes*; *gSubexpPtr*↑.*CompleteType*; *gSubexpPtr*↑.*CompleteClusterType*;
    KillSubexpression; *KillExpression*;
    end;
  gItemPtr↑.*FinishClusterType*;
  end

This code is used in section 1643.

1645. ⟨ Parse conditional registration cluster 1645 ⟩ ≡
begin *gItemPtr*↑.*FinishAntecedent*; *ReadWord*; *gItemPtr*↑.*CreateExpression*(*exAdjectiveCluster*);
gExpPtr↑.*CreateSubexpression*; *gSubexpPtr*↑.*StartAttributes*; *ATTSubexpression*(*lExpKind*);
if *lExpKind* ≠ *exAdjectiveCluster* **then**
begin *ErrImm*(*paAdjClusterExp*); **end**;
gSubexpPtr↑.*CompleteAdjectiveCluster*; *KillSubexpression*; *KillExpression*;
gItemPtr↑.*FinishConsequent*; *Accept*(*sy_For*, *paForExp*); *gItemPtr*↑.*CreateExpression*(*exType*);
gExpPtr↑.*CreateSubexpression*; *gSubexpPtr*↑.*StartType*; *gSubexpPtr*↑.*StartAttributes*;
GetAdjectiveCluster; *RadixTypeSubexpression*; *gSubexpPtr*↑.*CompleteAttributes*;
gSubexpPtr↑.*CompleteType*; *gSubexpPtr*↑.*CompleteClusterType*; *KillSubexpression*; *KillExpression*;
gItemPtr↑.*FinishClusterType*;
end

This code is used in section 1643.

1646. ⟨ Parse existential registration cluster 1646 ⟩ ≡
begin *gItemPtr*↑.*FinishConsequent*; *ReadWord*; *gItemPtr*↑.*CreateExpression*(*exType*);
gExpPtr↑.*CreateSubexpression*; *gSubexpPtr*↑.*StartType*; *gSubexpPtr*↑.*StartAttributes*;
GetAdjectiveCluster; *RadixTypeSubexpression*; *gSubexpPtr*↑.*CompleteAttributes*;
gSubexpPtr↑.*CompleteType*; *gSubexpPtr*↑.*CompleteClusterType*; *KillSubexpression*; *KillExpression*;
gItemPtr↑.*FinishClusterType*;
end

This code is used in section 1643.

1647. Reduction registration.

⟨ Parse definitions (`parser.pas`) 1621 ⟩ +≡
procedure *Reduction*;
var *lExpKind*: *ExpKind*;
begin *gBlockPtr*↑.*CreateItem*(*itReduction*); *ReadWord*;
if (*CurWord*.*Kind* = *Identifier*) ∧ (*AheadWord*.*Kind* = *sy_Arrow*) **then** *ErrImm*(*paFunctExp4*);
gItemPtr↑.*StartFuncReduction*; *TermExpression*; *gItemPtr*↑.*ProcessFuncReduction*;
Accept(*sy_To*, *paToExp*); *TermExpression*; *gItemPtr*↑.*FinishFuncReduction*; *Semicolon*; *Correctness*;
end;

1648. Identification registration.

⟨ Parse definitions (`parser.pas`) 1621 ⟩ +≡
procedure *Identification*;
begin *gBlockPtr*↑.*CreateItem*(*itIdentify*); *ReadWord*; { begin }
gItemPtr↑.*StartFuncIdentify*; *GetFuncPattern*; *gItemPtr*↑.*ProcessFuncIdentify*;
Accept(*sy_With*, *paWithExp*); *GetFuncPattern*; *gItemPtr*↑.*CompleteFuncIdentify*; { end; }
if *CurWord*.*Kind* = *sy_When* **then**
begin *ReadWord*;
repeat *gItemPtr*↑.*ProcessLeftLocus*; *Accept*(*Identifier*, *paIdentExp3*);
Accept(*sy_Equal*, *paEqualityExp1*); *gItemPtr*↑.*ProcessRightLocus*; *Accept*(*Identifier*, *paIdentExp3*);
until ¬*Occurs*(*sy_Comma*);
end;
Semicolon; *Correctness*;
end;

1649. Property registration.

⟨ Parse definitions (`parser.pas`) 1621 ⟩ +≡

```
procedure RegisterProperty;
  begin gBlockPtr↑.CreateItem(itPropertyRegistration);
  case PropertyKind(CurWord.Nr) of
  sySethood: begin ReadWord; Accept(sy_of, paOfExp); gItemPtr↑.StartSethoodProperties;
    TypeExpression; gItemPtr↑.FinishSethoodProperties; Justification;
    end;
  othercases begin SynErr(CurPos, paStillNotImplemented); end;
  endcases;
  Semicolon;
  end;
```

1650.

⟨ Parse definitions (`parser.pas`) 1621 ⟩ +≡

```
procedure RegistrationBlock;
  var DefPos: Position;
  begin gBlockPtr↑.CreateItem(itDefinition); gBlockPtr↑.CreateBlock(blRegistration);
  DefPos ← CurPos; ReadWord;
  while CurWord.Kind ≠ sy_End do
    begin StillCorrect ← 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: begin gBlockPtr↑.CreateItem(itLociDeclaration); ReadWord; FixedVariables; end;
      sy_Canceled: Canceled;
      sy_Case, sy_Suppose, sy_Per, sy_Hereby: UnexpectedItem;
      othercases PrivateItem;
      endcases;
      Semicolon;
      end;
    endcases;
    end;
  KillBlock; AcceptEnd(DefPos);
  end;
```

1651. Reservation.

⟨ Parse definitions (`parser.pas`) 1621 ⟩ +≡
procedure *Reservation*;
  begin *gBlockPtr*↑.*CreateItem*(*itReservation*); *ReadWord*;
  repeat *gItemPtr*↑.*StartReservationSegment*;
    repeat *gItemPtr*↑.*ProcessReservedIdentifier*; *Accept*(*Identifier*, *paIdentExp11*);
    until ¬*Occurs*(*sy_Comma*);
    Accept(*sy_For*, *paForExp*); *gItemPtr*↑.*CreateExpression*(*exResType*); *TypeSubexpression*;
    KillExpression; *gItemPtr*↑.*FinishReservationSegment*;
  until ¬*Occurs*(*sy_Comma*);
  gItemPtr↑.*FinishReservation*;
  end;

1652. Theorem.

⟨ Parse definitions (`parser.pas`) 1621 ⟩ +≡
procedure *Theorem*;
  begin *gBlockPtr*↑.*CreateItem*(*itTheorem*); *ReadWord*; *ProcessLab*; *gItemPtr*↑.*StartTheoremBody*;
  ProcessSentence; *gItemPtr*↑.*FinishTheoremBody*; *Justification*; *gItemPtr*↑.*FinishTheorem*;
  end;

1653. Axiom.

⟨ Parse definitions (`parser.pas`) 1621 ⟩ +≡
procedure *Axiom*;
  begin *gBlockPtr*↑.*CreateItem*(*itAxiom*); *ReadWord*; *ProcessLab*; *gItemPtr*↑.*StartTheoremBody*;
  ProcessSentence; *gItemPtr*↑.*FinishTheoremBody*; *gItemPtr*↑.*FinishTheorem*;
  end;

Section 24.8. SCHEME BLOCKS

1654.

⟨ Parse scheme block (`parser.pas`) 1654 ⟩ ≡
{ *Main (with Schemes)* }

```
procedure SchemeBlock;
  var SchemePos: Position;
  begin gBlockPtr↑.CreateItem(itSchemeBlock); gBlockPtr↑.CreateBlock(blPublicScheme); ReadWord;
  gBlockPtr↑.CreateItem(itSchemeHead); gItemPtr↑.ProcessSchemeName; SchemePos ← PrevPos;
  if CurWord.Kind = Identifier then ReadWord;
  ⟨ Parse scheme parameters 1655 ⟩;
  Accept(sy_RightCurlyBracket, paRightCurledExp3); gItemPtr↑.FinishSchemeHeading;
  Accept(sy_Colon, paColonExp3); FormulaExpression;  { Scheme-conclusion }
  gItemPtr↑.FinishSchemeThesis; ⟨ Parse scheme premises 1656 ⟩;
  gItemPtr↑.FinishSchemeDeclaration; ⟨ Parse justification for scheme 1657 ⟩;
  KillBlock;
  end;
```

This code is used in section 1493.

1655. ⟨ Parse scheme parameters 1655 ⟩ ≡

```
  Accept(sy_LeftCurlyBracket, paLeftCurledExp);
  repeat gItemPtr↑.StartSchemeSegment;
    repeat gItemPtr↑.ProcessSchemeVariable; Accept(Identifier, paIdentExp13);
    until ¬Occurs(sy_Comma);
    gItemPtr↑.StartSchemeQualification;
    case CurWord.Kind of
    sy_LeftSquareBracket: begin ReadWord; ReadTypeList; gItemPtr↑.FinishSchemeQualification;
      Accept(sy_RightSquareBracket, paRightSquareExp5);
      end;
    sy_LeftParanthesis: begin ReadWord; ReadTypeList; gItemPtr↑.FinishSchemeQualification;
      Accept(sy_RightParanthesis, paRightParenthExp9); Specification;
      end;
    othercases begin ErrImm(paWrongSchemeVarQual); gItemPtr↑.FinishSchemeQualification;
      Specification;
      end;
    endcases; gItemPtr↑.FinishSchemeSegment;
  until ¬Occurs(sy_Comma)
```

This code is used in section 1654.

1656. ⟨ Parse scheme premises 1656 ⟩ ≡

```
  if CurWord.Kind = sy_Provided then
    repeat gItemPtr↑.StartSchemePremise; ReadWord; ProcessLab; ProcessSentence;
      gItemPtr↑.FinishSchemePremise;
    until CurWord.Kind ≠ sy_And
```

This code is used in section 1654.

1657. ⟨ Parse justification for scheme 1657 ⟩ ≡

```
if CurWord.Kind = sy_Proof then
  begin KillItem;  { only KillItem which is run outside of Semicolon procedure }
  if ¬ProofPragma then
    begin gBlockPtr↑.StartSchemeDemonstration; IgnoreProof;
    gBlockPtr↑.FinishSchemeDemonstration;
    end
  else begin StillCorrect ← true; Accept(sy_Proof, paProofExp);
    gBlockPtr↑.StartSchemeDemonstration; LinearReasoning;
    if CurWord.Kind = sy_Per then NonBlockReasoning;
    AcceptEnd(SchemePos); gBlockPtr↑.FinishSchemeDemonstration;
    end;
  end
else begin Semicolon;
  if ¬ProofPragma then
    begin gBlockPtr↑.StartSchemeDemonstration; IgnoreProof;
    gBlockPtr↑.FinishSchemeDemonstration;
    end
  else begin StillCorrect ← true;
    if CurWord.Kind = sy_Proof then
      begin WrongWord(paProofExp); StillCorrect ← true; ReadWord;
      end;
    gBlockPtr↑.StartSchemeDemonstration; LinearReasoning;
    if CurWord.Kind = sy_Per then NonBlockReasoning;
    AcceptEnd(SchemePos); gBlockPtr↑.FinishSchemeDemonstration;
    end;
  end
```

This code is used in section 1654.

Section 24.9. MAIN PARSE PROCEDURE

1658. The main *Parse* method essentially skips ahead to the first "`begin`", then skips ahead to the first top-level block statement.

define *skip_to_begin* ≡ *ReadTokenProc*;
  while (*CurWord.Kind* ≠ *sy_Begin*) ∧ (*CurWord.Kind* ≠ *EOT*) **do** *ReadTokenProc*

⟨ Main parse method (`parser.pas`) 1658 ⟩ ≡
procedure *Parse*;
  begin *skip_to_begin*; { Skips ahead until EOT or finds 'begin' }
  if *CurWord.Kind* = *EOT* **then** *ErrImm*(213)
  else ⟨ Parse proper text 1659 ⟩; { *CurrWord.Kind* = *sy_Begin* }
  KillBlock;
  end;

This code is used in section 1493.

1659. Parsing the "text proper" checks that we have encountered a "`begin`" keyword, then parses the block statements in the article's contents.

Note that *ProcessBegin* (§1214) and *StartProperText* (§1216) are both implemented in the extended block class.

⟨ Parse proper text 1659 ⟩ ≡
  begin *gBlockPtr*↑.*StartProperText*; *gBlockPtr*↑.*ProcessBegin*; *Accept*(*sy_Begin*, 213);
  while *CurWord.Kind* ≠ *EOT* **do** ⟨ Parse next block 1660 ⟩;
  end

This code is used in section 1658.

1660. When parsing the next top-level block in a Mizar article, we tell Mizar's parser we are not in "panic mode". Then we test for unexpected "`end`" tokens. If we can recover a "`begin`" token, just start the loop over again.

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.

⟨ Parse next block 1660 ⟩ ≡
  begin ⟨ Parse pragmas and begins 1661 ⟩;
  StillCorrect ← *true*; { we are not in panic mode }
  if *CurWord.Kind* = *sy_End* **then**
    begin ⟨ Skip all `end` tokens, report errors 1662 ⟩;
    if *CurWord.Kind* = *sy_Begin* **then** *continue*;
    end;
  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 }
  end

This code is used in section 1659.

1661. The *ProcessPragmas* (§1585) consumes a token when the current token is a pragma. So we effectively have a loop where we consume all the pragmas and the "`begin`" keywords until we find something else.

⟨ Parse pragmas and begins 1661 ⟩ ≡
  while *CurWord.Kind* ∈ [*sy_Begin*, *Pragma*] **do**
    begin *ProcessPragmas*;
    if *CurWord.Kind* = *sy_Begin* **then**
      begin *gBlockPtr*↑.*ProcessBegin*; *ReadTokenProc*;
      end;
    end

This code is used in section 1660.

1662. In the unfortunate event that the parser has stumbled across an "`end`" token, skip all the "`end`" and semicolon tokens and report errors.

⟨ Skip all `end` tokens, report errors 1662 ⟩ ≡
  repeat *ErrImm*(216); *ReadTokenProc*;
    if *CurWord.Kind* = *sy_Semicolon* **then** *ReadTokenProc*;
  until *CurWord.Kind* ≠ *sy_End*

This code is used in section 1660.

1663. Deferred.

This will not be analyzed until `first_identification.pas`

⟨ Class for Within expression 1663 ⟩ ≡

This code is used in section 737.

1664. ⟨ Within expression AST implementation 1664 ⟩ ≡

This code is used in section 736.

1665. Index. Underlined entries in an index item refers to which section defines the identifier. Primitive types (*char*, *Boolean*, *string*, etc.) are omitted from the index.

⟨ Parse pragmas and begins 1661 ⟩ Used in section 1660.
⟨ Parse predicate pattern 1613 ⟩ Used in section 1612.
⟨ Parse private formula 1550 ⟩ Used in section 1549.
⟨ Parse proper text 1659 ⟩ Used in section 1658.
⟨ Parse reference to scheme from MML 1575 ⟩ Used in section 1574.
⟨ Parse scheme block (`parser.pas`) 1654 ⟩ Used in section 1493.
⟨ Parse scheme parameters 1655 ⟩ Used in section 1654.
⟨ Parse scheme premises 1656 ⟩ Used in section 1654.
⟨ Parse segment of fixed variables 1566 ⟩ Used in section 1565.
⟨ Parse semicolon-separated items in a notation block 1640 ⟩ Used in section 1639.
⟨ Parse simple Fraenkel expression or "the set" 1517 ⟩ Used in section 1515.
⟨ Parse simple justifications (`parser.pas`) 1571, 1574, 1576 ⟩ Used in section 1493.
⟨ Parse single reference 1572 ⟩ Used in section 1571.
⟨ Parse start of XML tag 509 ⟩ Used in section 508.
⟨ Parse statement of linear reasoning 1587 ⟩ Used in section 1586.
⟨ Parse statements and reasoning (`parser.pas`) 1577, 1578, 1579, 1580, 1585, 1586, 1591, 1598, 1599 ⟩ Used in section 1493.
⟨ Parse structure as radix type 1527 ⟩ Used in section 1525.
⟨ Parse the fields of the structure definition 1619 ⟩ Used in section 1615.
⟨ Partial Binary integer Functions 430 ⟩ Used in section 183.
⟨ Partial integer function implementation 360 ⟩ Used in section 183.
⟨ Partial integers to Pair of integers Functions 448 ⟩ Used in section 183.
⟨ Pattern objects (`wsmarticle.pas`) 933, 935, 937, 939, 941 ⟩ Used in section 930.
⟨ Placeholder term (abstract syntax tree) 758 ⟩ Used in section 755.
⟨ Pop **skip**s remaining cases 1251 ⟩ Used in section 1250.
⟨ Pop a "`let`" statement 1263 ⟩ Used in section 1250.
⟨ Pop a conclusion or regular statement 1261 ⟩ Used in section 1250.
⟨ Pop a definition item 1264, 1265, 1266, 1267, 1270 ⟩ Used in section 1250.
⟨ Pop a proof step 1254, 1255, 1257, 1259 ⟩ Used in section 1250.
⟨ Pop a registration item 1272, 1275 ⟩ Used in section 1250.
⟨ Pop a token from the underlying tokens stack 630 ⟩ Used in sections 629 and 629.
⟨ Populate global variables with XML entities 1009 ⟩ Used in section 1008.
⟨ Populate the current line 636 ⟩ Used in section 635.
⟨ Predicate declarations for arbitrary-precision arithmetic 133 ⟩ Used in section 128.
⟨ Prepare to read in the contents of XML file 496 ⟩ Used in section 495.
⟨ Private functor term (abstract syntax tree) 768 ⟩ Used in section 755.
⟨ Process "`case`" (local procedure) 1592 ⟩ Used in section 1591.
⟨ Process attributes (`parser.pas`) 1524 ⟩ Used in section 1523.
⟨ Process miscellany (`parser.pas`) 1559, 1560, 1561, 1562, 1563, 1564, 1565, 1567, 1568, 1569, 1570 ⟩ Used in section 1493.
⟨ Process post-qualified segment 1507 ⟩ Used in section 1506.
⟨ Public constants for `dicthan.pas` 555 ⟩ Used in section 554.
⟨ Public constants for `syntax.pas` 682 ⟩ Used in section 688.
⟨ Public declaration for Stream Object 520 ⟩ Used in section 515.
⟨ Public declaration for Text Stream Object 524 ⟩ Used in section 515.
⟨ Public declaration for XML Output Stream 533 ⟩ Used in section 515.
⟨ Public function declarations for `dicthan.pas` 557 ⟩ Used in section 554.
⟨ Public interface for MScanner 720, 722, 723 ⟩ Used in section 719.
⟨ Public interface for XML Input Stream 528 ⟩ Used in section 515.
⟨ Public interface for `mobjects.pas` 184, 185, 186, 191, 194, 195, 219, 235, 250, 267, 281, 286, 295, 299, 307, 309, 340, 341, 350, 359, 379, 397, 402, 418, 429, 447, 461 ⟩ Used in section 182.
⟨ Public procedures for `syntax.pas` 689 ⟩ Used in section 688.

⟨ Try to synchronize after failing to find initial '`case`' or '`suppose`' 1594 ⟩ Used in section 1591.
⟨ Tuples of integers 342 ⟩
⟨ Type declarations for XML I/O 515 ⟩ Used in section 514.
⟨ Type declarations for `xml_parser.pas` 489 ⟩ Used in section 487.
⟨ Type declarations for scanner 589 ⟩ Used in section 588.
⟨ Types for arbitrary-precision arithmetic 130 ⟩ Used in section 128.
⟨ Update content of *nLastWSItem* based on type of item popped 1250 ⟩ Used in section 1249.
⟨ Update max arguments for structure symbol, if needed 1618 ⟩ Used in section 1615.
⟨ Variable (abstract syntax tree) 738 ⟩ Used in section 737.
⟨ Variable AST constructor 739 ⟩ Used in section 736.
⟨ Variables for finishing a long term in a subexpression 1426 ⟩ Used in section 1417.
⟨ Variables for slicing a phrase 612, 615, 618, 621, 623, 625 ⟩ Used in section 613.
⟨ Vocabulary object declaration 573 ⟩ Used in section 556.
⟨ Weakly strict Item class 860 ⟩ Used in section 856.
⟨ Whoops! ID turns out to be invalid, insert an error token, then continue 617 ⟩ Used in section 616.
⟨ Whoops! Identifier turned out to be a number! 626 ⟩ Used in section 622.
⟨ Whoops! We found an unknown token, insert a 203 error token 627 ⟩ Used in section 613.
⟨ Windows implemenation for *CtrlSignal* 99 ⟩ Used in section 96.
⟨ Windows implemenation for *InitCtrl* 96 ⟩ Used in section 94.
⟨ Within expression AST implementation 1664 ⟩ Used in section 736.
⟨ Write symbols to vocabulary XML file 587 ⟩ Used in section 585.
⟨ Write vocabulary counts to XML file 586 ⟩ Used in section 585.
⟨ Zero and units for arbitrary-precision 131 ⟩ Used in section 128.
⟨ _format.pas 647 ⟩
⟨ "It" term (abstract syntax tree) 792 ⟩ Used in section 755.
⟨ abstract_syntax.pas 735 ⟩
⟨ dicthan.pas 554 ⟩
⟨ errhan.pas 102 ⟩
⟨ implementation of mizenv.pas 26, 28, 31, 35, 46 ⟩ Used in section 23.
⟨ info.pas 485 ⟩
⟨ interface for mizenv.pas 24, 27, 30, 34 ⟩ Used in section 23.
⟨ librenv.pas 463 ⟩
⟨ mconsole.pas 54 ⟩
⟨ mizenv.pas 23 ⟩
⟨ mobjects.pas 182 ⟩
⟨ monitor.pas 89 ⟩
⟨ mstate.pas 83 ⟩
⟨ mtime.pas 115 ⟩
⟨ numbers.pas 128 ⟩
⟨ parser.pas 1492 ⟩
⟨ parseraddition.pas 1199 ⟩
⟨ pcmizver.pas 77 ⟩
⟨ pragmas.pas 1193 ⟩
⟨ scanner.pas 588, 719 ⟩
⟨ syntax.pas 681 ⟩
⟨ wsmarticle.pas 850 ⟩
⟨ xml_dict.pas 462 ⟩
⟨ xml_inout.pas 514 ⟩
⟨ xml_parser.pas 487 ⟩
⟨ *DisplayLine* global constant 58 ⟩ Used in section 57.
⟨ *IntSequence* implementation 403 ⟩ Used in section 183.
⟨ *IntSet* Implementation 419 ⟩ Used in section 183.

Mizar Parser